Addressing the Knowledge Transfer Problem in Secure Software Development Through Anti-Patterns

A thesis submitted for the degree of Doctor of Philosophy (PhD)

by

Tayyaba Nafees

School of Design and Informatics, Division of Cybersecurity, Abertay University.

February, 2019

Declaration

Candidate's declarations:

I, Tayyaba Nafees, hereby certify that this thesis submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy (PhD), Abertay University, is wholly my own work unless otherwise referenced or acknowledged. This work has not been submitted for any other qualification at any other academic institution.

Signed [candidates signature]…….
Date……..

Supervisor's declaration:

I, Natalie Coull, hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in Abertay University and that the candidate is qualified to submit this thesis in application for that degree.

Signed [Principal Supervisors signature]……...
Date……..

Certificate of Approval

I certify that this is a true and accurate version of the thesis approved by the examiners, and that all relevant ordinance regulations have been fulfilled.

Supervisor……..…….
Date……..

Acknowledgements

"[All] praise is [due] to Allah, Lord of the worlds"

First and foremost, I'd like to thank my supervisory team Dr Natalie Coull, Dr Ian Ferguson and Dr Adam Sampson for providing me constant guidance and support throughout the duration of the research and the write-up. I'd like to express my gratitude to Sascha Roschy for his excellent guidance and expertise that helped shape the thesis into its finished format. It goes without saying that this thesis wouldn't have happened without you, so I thank you for giving me confidence in my writing skills.

Thanks to Dr Nia White, for supporting and encouraging me to purse my dream as an entrepreneur. I hope that we will be able to continue working together in the future.

Thanks to Gameelah Gahfoor, for helping me in running my experimental study.

To my Dad and Mom (Ammi), thank you for being always with me. Thanks to all my siblings, especially Babar, Khawar and Dr Rabia., for always being there to pay my credit card shopping bills. But more importantly, thank you all for supporting me during the PhD process and for your patience with how long it's taken to finish. To Aden, for always being there to listen my nonsense conversations and making me laugh even from the deepest depths of thesis writing.

To all the people I have met along the way- I came to Abertay as a postgraduate student. I've had an amazing time here, but it would not have been the same without these people. Wendy, for constant advice, Salma, for support and encouragement, Alice, for love and care, and all my fellow squash friends. Thank you all for wee lunch breaks, long walks with laughter and tiring squash games to clear my mind.

Abstract

There is a distinct communication gap between software engineering and cybersecurity communities when it comes to addressing reoccurring security problems, known as vulnerabilities. Many vulnerabilities are caused by software errors that occur due to developers' common mistakes. Insecure software development practices are common due to a variety of factors, which include inefficiencies within existing knowledge transfer mechanisms based on vulnerability databases (VDBs) and pattern-based approaches, software developers perceiving security as an afterthought, and lack of consideration of security as part of the Software Development Lifecycle (SDLC). The resulting communication gap also prevents developers and security experts from successfully sharing essential security knowledge.

This thesis identifies the major issues in the transfer of vulnerability knowledge (vulnerability databases (VDBs)) using the existing pattern-based approaches, which prohibits developers from finding causes of vulnerabilities (errors) and mitigating them; Experts of both domains struggle to understand each other's security perspectives due to lack of understanding and sharing of common terms, languages and procedures.

To address these issues, a hybrid pattern-based approach, Vulnerability Anti-pattern (VAPs), has been developed consisting of two types that encapsulates knowledge of existing vulnerabilities to bridge the communication gap between security experts and software developers. A catalogue of VAPs based on the most commonly occurring vulnerabilities has been created that assists software developers in developing an awareness of how malicious hackers can exploit errors in software.

The evaluation was performed through a series of experimental studies to measure the effectiveness of VAP in order to raise awareness of poor security practices that lead to vulnerabilities. Whilst the results indicate the improvement of developers' awareness of vulnerabilities and encouraging them to create secure software systems.

Table of Contents

List of Tables

List of Figures

Definitions and Acronyms

Term	Definition
Anti-Pattern	An anti-pattern describes a “general form, the primary causes which led to the general form; symptoms describing how to recognise the general form; the consequences of the general form; and a refactored solution describing how to change the Anti-Pattern into a healthier situation”(Brown et al. 1998).
Exploit/ Misuse	A “technique to breach the security of a network or information system in violation of a security policy” (Committee on National Security Systems (CNSS) 2003).
Vulnerability	A “weakness in a system, application, or network that is subject to exploitation or misuse” (Kissel 2013).
Software Fault Pattern	The Software Fault Patterns (SFP) are a clustering of CWEs into related weakness categories. Each cluster is “factored into formally defined attributes, with sites (footholds), conditions, properties, sources, sinks, etc. This work overcomes the problem of combinations of attributes in CWE” (Mansourov 2011).
Penetration Tester	A tester who is “used to test the external perimeter security of a network or facility to find vulnerabilities that an attacker could exploit” (CERT 2015).

VDB	Vulnerability Database
VAP	Vulnerability Anti-Pattern
SP	Security Pattern
CWE	Common Weakness Enumeration
CVE	Common Vulnerabilities and Exposures
CAPEC	Common Attack Pattern Enumeration and Classification
SDLC	Software Deployment Lifecycle
SFP	Software Fault Pattern

1 Introduction

There is a distinct communication gap between the software engineering and cybersecurity communities when it comes to addressing the class of reoccurring security problems known as vulnerabilities. Software errors made by software developers cause many vulnerabilities. Insecure software development practices are common due to a variety of factors, which include inefficiencies within existing knowledge transfer mechanisms based on vulnerability databases (VDBs) such as CAPEC and CWE, software developers perceiving security as an afterthought, and a lack of consideration of security as part of the Software Development Lifecycle (SDLC). The resulting communication gap also prevents developers and security experts from successfully sharing essential security knowledge. The cybersecurity community makes their expert knowledge available in various forms including vulnerability databases and security-related pattern catalogues such as Security Patterns, Anti-Patterns, and Software Fault Patterns. However, these existing sources are not effective at providing software developers with an understanding of how malicious hackers can exploit vulnerabilities in the software systems they create. This is due to the complex structure of existing vulnerability knowledge sources coupled with a lack of understanding of how to use them during the SDLC (Van and McGraw 2005, McGraw 2012, Yun-hua and Pei 2010).

For example, Jafari and Rasoolzadegan (2016) work on securing Gang of Four (GoF) design patterns; however, it raised the concerns related to adding overhead against increased security. Dougherty et al (2009) extension of work lacks the information related to potential vulnerabilities and solutions and of how to tackle them. This suggests that improved use of existing security-related patterns embedded with vulnerability knowledge can help to improve the security of software. As developers are familiar with pattern-based approaches, the use of Vulnerability Anti-Patterns (VAPs) to transfer vulnerability knowledge to developers in a usable way is proposed.

1.1 Vulnerability Anti-Pattern

The Vulnerability Anti-Pattern is a hybrid solution, which encapsulates knowledge of vulnerabilities from VDBs and presents this knowledge to developers so that they can understand how poor coding software practices can be exploited. This increased understanding and awareness of malicious hackers' techniques will contribute to the

development of more secure software and aid developers' understanding of the prevention of software vulnerabilities.

The primary contribution of this thesis is twofold:

1) A new pattern-based approach – the Vulnerability Anti-Pattern –that encapsulates knowledge of existing vulnerabilities to bridge the communication gap between security experts and software developers.
2) The use of a Vulnerability Anti-Patterns (VAPs) catalogue to provide information about the most commonly occurring vulnerabilities that software developers can use to learn how malicious hackers can exploit errors in code.

1.2 Difference between the VAP and Existing Security Related Pattern-Based Approaches

The Vulnerability Anti-Pattern template is based on that proposed in Brown et al (1998). However, existing anti-patterns are not intended to capture relationships between poor practices and vulnerabilities, and do not provide mechanisms for capturing cybersecurity domain knowledge.

We argue that existing pattern-based techniques – security patterns (Steel and Nagappan 2006), software fault patterns (Mansourov 2011) and attack patterns (MITRE Corporation 2014) – are ineffective at capturing and transferring necessary knowledge of vulnerabilities. Anand, Ryoo, and Kazman (2014) and Hafiz (2011) report that security patterns are harder for developers to use than conventional design patterns. Dimitrov (2016) finds that the structure and semantics of software fault patterns (SFPS) do not adequately capture all classes of vulnerabilities, and do not align well with existing formal notations used by software engineers. In addition, NIST reports that SFPs, as used in the CWE database, do not appropriately describe the causes or consequences of related vulnerabilities (Black 2017). Faily, Parkin and Lyle (2014) evaluated the use of both security patterns and attack patterns within the software development process and found problems with the identification of specific vulnerabilities in the system, and the complex interactions between security and attack patterns; they report that there is "a dearth of work" evaluating the use of attack patterns by software engineers. The intended audience for security patterns is security experts rather than developers (Bunke 2015; Fernandez-Buglioni 2013), and the need for usable and accessible knowledge about vulnerabilities is highlighted by (Van and McGraw 2005, Fahl et al. 2013, Acar et al. 2016).

As Vulnerability Anti-Patterns (VAPs) are intended to capture recurring errors that lead to vulnerabilities, we extended the template to include knowledge extracted from vulnerability databases to make it understandable to software developers. As an example of VAP, the “Use of Potentially Dangerous Function”:
Anti-Pattern describes the use of functions within a software system that are likely to result in exploitable behaviour when a safer alternative is available. An instance of this anti-pattern is the use of C’s *strcpy* function, which provides no inherent safeguards against incorrect source or target buffer sizes, frequently resulting in faults such as buffer-overflow vulnerabilities (Howard and Lipner 2011). Vulnerabilities resulting from improper use of *strcpy* are so common – as evidenced in the CVE database (MITRE Corporation 2016) – that some standards prohibit its use entirely (OWASP 2015).

The “Use of Potentially Dangerous Function” is proposed as a corresponding solution, such as safe library functions *strcpy*. This VAP captures security expert knowledge extracted from these sources in a form that is understandable for software developers.

1.3 Background

Information technology (IT) infrastructures have evolved over the last two decades and are now integrated into virtually every aspect of our lives. The software is a component of the IT systems that form a large part of the present day. For example, the UK is considered one of the world’s leading digital societies (U.K National Crime Agency 2017). This digital transformation of society, however, somehow creates a new set of dependencies such as e-banking, social networking and e-commerce. The massive expansion of the Internet beyond traditional computers and mobile phones fashions a new electronic medium of digital networks, which is called “Cyberspace”, used to store, modify and communicate information (Blackwell and Zhu 2014b). This also includes other information systems that support businesses, infrastructures and services such as power grids, air traffic control systems, satellites, medical technologies and industrial plants.

The term “cybersecurity” has been coined to refer to the protection of assets that are directly or indirectly connected to the Internet (CERT 2015). There will always be attempts to exploit weaknesses to launch cyber-attacks. This threat cannot be eliminated completely, but the risk can be significantly reduced to a level that allows users to continue to prosper, and benefit from the potential opportunities that digital

technology brings (Mansourov and Campara 2010). In terms of economic investment to maintain cyberspace security, the recent National Cyber Security Strategy 2017-2021 (National Cyber Security Centre 2016) estimated that the UK Government invests £1.9 billion in cybersecurity (U.K National Crime Agency 2017). In a comparatively recent article (CERT 2015), both cybersecurity experts and software engineers noted that people's critical dependency on the Internet had changed so radically since the field's inception. This has created a mind shift in the way we define as its early definition is limited to (i) computer security, and (ii) internet security. However, the modern definition of cybersecurity further includes process, controls and technologies that are designed to protect systems, networks and data from cyber-attacks (Craigen, Diakun-Thibault and Purse 2014).

Recently, there has been renewed interest in pattern-oriented research methods whose primary focus is to study a subject domain such as Cyberspace by identifying the potential relationship between different types of cyber-patterns. Uses the term "Cyber-patterns" to refer to unifying design patterns with security and attack related patterns in cyberspace. There is little consensus about what cyberspace actually means; however, Morningstar and Farmer (2003) have observed that cyberspace is defined more by the social interactions involved rather than its technical implementation.

Either way, software systems and the Internet are intertwined with each other in cyberspace where users interact for social, information and creative purposes. As software systems are continuously growing in size, complexity and connectivity, the growing risks are related to their malicious use (McGraw 2006). For malicious attackers, the attack surface[1] has been increasing by 70% over the last decade, which contributes to the misuse of software systems and the exploitation of vulnerabilities (Choo 2011).

Programmers can make mistakes during the development process, which could generate or lead to software vulnerabilities. A software vulnerability is a flaw or defect in the software system that can be exploited by an attacker in order to obtain some privileges in the system. It means the vulnerabilities offer possible entry points to the system. Despite the knowledge about vulnerabilities nowadays, there is still a growing tendency in the number of reported vulnerabilities, and this is the reason why software

[1] The total sum of the vulnerabilities in a given computing device or network that are accessible to a hacker.

security has become an important field of research. The presence of vulnerabilities in software makes it necessary to have techniques that can help and assist developers in understanding and mitigating errors during the development of the software.

In this chapter, the research problem upon which this thesis is based is introduced, i.e. why the development of mistakes or errors that lead to vulnerabilities are posing a significant challenge for developers in the creation of secure software. The research aims to address the existing communication gap between software engineers and cybersecurity experts when it comes to addressing reoccurring security problems. The research proposed questions that contribute to bridging the knowledge gap between both communities. Based on these questions, the main claim made by this thesis is to justify the research approach which is derived via the Vulnerability Anti-Pattern (VAP). This chapter concludes by presenting an overview of the dissertation in Section 1.4, followed by its detailed structure in Section 1.9 and publications arise from thesis work in Section 1.10.

The key objectives of the chapter are:

- Provide an overview of the thesis in section 1.4.
- Describe the thesis statement and arising research questions in sections 1.5, 1.6, 1.7 and 1.8.
- Present the dissertation structure and published work in sections 1.9 and 1.10.

1.4 Thesis Motivation (Story Line)

Security is a challenging task for software developers. The development process of most software systems prioritises efficiency, cost and user convenience, deadlines, but does not always have security designed in from the start. Thus, while users' expectations for technology innovation continue to increase, the quality of software security and security often falls short. The unfortunate reality, however, is that software developers struggle against these recurring and consistent software errors (commonly known as vulnerabilities), e.g. buffer overflows and integer overflows are exploited by hackers on a daily basis (OWASP 2015). The Software can be made vulnerable through a variety of factors ranging from complex requirements specifications, accidental introduction of software errors, and the adoption of poor software engineering practices. Malicious hackers may have the capabilities and motivations to take advantage of these simple mistakes, which may be unknown to the developer who inadvertently introduced them. Furthermore, a lack of understanding of what

vulnerabilities are and how they can be exploited may be the reason for developers to continue making the same errors repeatedly during the development process.

The frequency of recurrence of recently discovered vulnerabilities in CVE (MITRE Corporation 2015a) shows that malicious hackers know a lot more about attacking systems than the developers who created them, indicating that the effectiveness of attackers can be traced back to their extensive knowledge sharing as discussed further in literature review chapter. In comparison, security experts and software developers fail to show similar efficiency in their knowledge sharing (Mouratidis, Giorgini and Manson 2003, Yun-hua and Pei 2010, McGraw 2012). Although the problem of frequently recurring software vulnerabilities is very well known, no standard solution has been universally adopted (Mansourov and Campara 2010). Furthermore, according to the National Cyber Security Centre (NCSC) “in 2016, cybercrime cost the UK economy £29.1 billion and is growing at an alarming rate” (Levi et al. 2016). As a corollary, users would also like software systems to be as secure as they are usable. However, this presents a challenging task for software developers due to an augmented complex network of people and software systems against vulnerabilities.

The question is: “Why do developers repeatedly make security mistakes?”

In general, software developers do not thoroughly understand the security issues and their main focus is usually delivering features and functionalities, rather than making sure that software is secure (Howard 2004). Moreover, in the software engineering community, the common trend is to tackle security during the late stages of the Software Development Lifecycle (SDLC). Ironically, this is estimated to be 30 times more expensive than considering security in the early stages of the SDLC (Howard 2004). The problem arises when considering how to bridge the distinct communication gap between software engineering and cybersecurity experts as part of a resilient system development process. Software engineers (Shiralkarand 2009) recommend that security should be implemented from the early stages of development and should be considered as important as other functional requirements. In the same manner, cybersecurity experts suggest the early use of security domain knowledge during software development. There are, however, many reasons why this might not occur in practices:

- Developers face limitations with regards to time and finances (Hans 2010).

- There is a lack of knowledge sharing between both experts (developers and security) (Yun-hua and Pei 2010, McGraw 2012).
- Developers present no threat. However, to build secure software systems, there is a need to provide developers with usable and understandable security knowledge (Green and Smith 2016, Acar et al. 2016, Witschey et al. 2015).

Evidence from previous studies reflects that there is a knowledge gap between developers and security experts, which requires special attention in order to deal with vulnerabilities (Ghani et al. 2013, Xie, Lipford, and Chu 2011, Fahl et al. 2013, Acar et al. 2016). Due to this communication gap, the understanding of how common errors in software development result in exploitable vulnerabilities in software systems is limited (Morgan 2016). For example, in 2016, 10626 Cross-Site Scripting (XSS) vulnerability exploitations were confirmed by CVE (MITRE Corporation 2015a) and considered one of the most prevalent web application security threat (Wichers 2017). Despite XSS prevalence and severe consequences, generally, XSS mitigation only requires a simple input validation during development. This has revealed that XSS is one of the most common types of vulnerability in web applications, yet many developers remain unaware of it and unable to identify instances due to their lack of understanding of existing complicated cybersecurity knowledge.

This thesis argues that an easily understood (usable) representation of recurring exploitable development errors can help developers' awareness of how malicious hackers exploit these errors. Thus, it would be fruitful to study failures, identify the recurring poor software practices and suggest solutions to these problems. This concept is known as a negative pattern or an anti-pattern. This study proposes a new pattern-based approach based upon the improved use of: "the Vulnerability Anti-Pattern" that encapsulates knowledge of existing vulnerabilities as a solution to bridge the gap between cybersecurity experts and software developers. A catalogue of Vulnerability Anti-Patterns (VAPs), based on the most commonly occurring vulnerabilities, is developed that software developers can use to learn how malicious hackers exploit errors and address these during SDLC.

1.5 Rationale for Research

Despite the software engineering community's endeavours, and the cybersecurity community's best efforts, the number of serious software exploitations is increasing: According to IBM X-Force Threat Intelligence Index report (2017) tracked 10,197

software vulnerabilities in 2016 (Alvarez et al. 2017). Generally, developers are not trained on how to leverage existing vulnerability database sources in order to learn how to avoid development errors that potentially cause vulnerabilities. Insecure software development practices are common due to a variety of factors, which include (i) inefficiencies within existing knowledge transfer mechanisms based on vulnerability databases (VDBs) and pattern-based approaches, (ii) software developers perceiving security as an afterthought, and (iii) lack of consideration of security as part of the Software Development Lifecycle (SDLC). This information gap between software developers and cybersecurity experts has directly led to widespread software vulnerabilities. The frequency of recurrence of commonly discovered vulnerabilities in databases, such as CVE, confirms that software developers make the same errors repeatedly during the development process. Consequently, it would be fruitful to study failures, identify recurring poor software practices and find solutions for these problems (Busch, Koch and Wirsing 2014, McGraw 2004, Howard and Lipner 2009). In addition to investigating how cybersecurity knowledge sources could be used to bridge the understanding gap of security information for software developers.

1.6 Thesis Statement

The essential argument made in this thesis is that using a pattern-based approach to the documentation and communication of knowledge about recurring security mistakes (and their amelioration) made during the process of software development can lead to the production of more secure software. The need for such an approach is based on the observation that ‘vulnerability knowledge’ is not currently transferred effectively between the disparate cybersecurity and software development communities. Whilst existing pattern-based approaches have been shown to improve developers’ understanding of vulnerabilities, it is further hypothesised that constraints in the design/style of patterns used have limited the effectiveness of knowledge transfer in current approaches.

This thesis statement will be defended through work which seeks to answer the following research questions.

1.7 Research Questions (RQ)

The work described in this thesis is thus based on proving the research question: “Can a pattern-based approach (Vulnerability Anti-Pattern) be effective in bridging the security knowledge gap between software developers and security experts in order to help developers in the creation of secure software systems?”

This research question can be broken down into the following questions:

- **RQ1:** Do software developers have an effective understanding of errors that lead to the creation of vulnerabilities, coupled with an awareness of how malicious hackers can exploit these errors?
- **RQ2:** Why are current attempts, in the form of patterns and catalogues of vulnerabilities, not successful in communicating security knowledge to software developers?
- **RQ3:** Do developers know how to mitigate these recurrent errors during the Software Development Lifecycle (SDLC)?

This work is original in that no previous study has used the Anti-Patterns construct to capture vulnerabilities to educate developers against common vulnerabilities to facilitate the creation of secure software.

1.8 Research Hypotheses (RH)

This research posits the following hypotheses:

- **RH1:** Software developers cannot radically recognise recurring software vulnerabilities during the Software Development Lifecycle (SDLC).
- **RH2:** Current attempts, in the form of patterns and catalogues of vulnerabilities, are generally not successful in communicating security knowledge to software developers.
- **RH3:** Anti-Patterns can provide sufficient awareness and understanding of vulnerabilities in order to enable developers to create more secure software.

1.9 Dissertation Overview

This dissertation is broken down into the subsequent nine chapters. Chapters 2 and 3 situate the dissertation and present the existing pattern-based approaches used to derive Vulnerability Anti-pattern (VAP). Chapters 4, and 5 present the overview of VAP design and create a catalogue of VAPs, which is validated by the pilot and industrial

studies described in Chapters 6, 7, and 8. Finally, Chapters 9, and 10 discuss the results and review the thesis.

2. Literature Review

Chapter 2 provides an overview of the on temporary state-of-affairs in the design of secure software development practices. This is achieved by describing the changing trends in cybersecurity that increase the importance of secure software development. Existing pattern-based approaches proposed by the cybersecurity and software engineering communities are evaluated to alleviate the severity of security problems during software development. Given the dissertation's focus an in-depth study of several pattern-based software engineering & cybersecurity approaches are considered from a security perspective. The analysed pattern-based approaches are security patterns, attack patterns, software fault patterns, and anti-patterns. The literature review has emphasised the fact that these existing efforts from both communities are not effective in terms of providing essential awareness about software development errors that create knowledge gaps. The chapter concludes with a brief evaluation of why security is a growing problem for developers due to existing knowledge gaps between software engineers and cybersecurity experts.

3. Criticism of Existing Pattern-Based Approaches and the Derivation of a New Approach: Vulnerability Anti-Pattern (VAP)

Chapter 3 describes the proposed methodology used to validate and achieve research propositions. It critiques the existing pattern-based approaches taken by the cybersecurity and software engineering communities, before presenting the derived research method employed. At the end of this chapter, the derivation of Vulnerability Anti-Pattern (VAP) process is proposed as improved use of the anti-pattern approach. VAP contributes to raising the awareness of developers about common vulnerabilities that is necessary for the development of secure software.

4. Vulnerability Anti-Patterns: A Timeless Way to Capture Poor Software Practices (Vulnerabilities)

Chapter 4 presents a proposed hybrid solution "Vulnerability Anti-Pattern", which encapsulates vulnerability knowledge from vulnerability databases (VDBs) and presents this knowledge to developers so that they can understand how malicious hackers can exploit poor software practices. This chapter provides the VAP definition and conceptual design. A justification is presented on how VAPs can help fill the knowledge gap between communities – software engineering and cybersecurity – by encapsulating knowledge of commonly occurring vulnerabilities. This is followed by

the solution presented by this research; merging cybersecurity knowledge into anti-patterns to generate a new pattern-based approach in the form of VAPs. Two types of VAPs are proposed; formal and informal, in terms of how the vulnerabilities are addressed. Collectively, this pattern-based approach helps developers to understand and become aware of malicious hackers' techniques and contributes to the development of more secure software. This chapter justifies the related concepts such as a vulnerability, an anti-pattern and their relationship. This chapter ends with a conclusion to describe both the proposed templates of VAPs: formal and informal.

5. Creating a Catalogue of Vulnerability Anti-Patterns (VAPs)

Chapter 5 presents the catalogue of Vulnerability Anti-Patterns (VAPs) including 12 vulnerabilities chosen from the OWASP list of "Top 25 Most Dangerous Software Errors". This chapter presents the clustering approaches to develop a catalogue of VAPs. The chapter concludes by describing both catalogues of VAPs: formal and informal.

6. Pilot Study -I (PS-I)

Chapter 6 is the part of the evaluation process, which describes the quantitative analysis of pilot study-I (without intervention) and its results. This chapter elucidates the pilot-study-I, including detail of participants and statistical analysis of data. The chapter concludes with statistically significant outcomes of the study.

7. Pilot Study -II (PS-II)

As a part of the evaluation process, chapter 7 describes the quantitative analysis of pilot-study-II and its results. This chapter explains the experiment design, its participants and analysis of the statistical results. The chapter concludes with significant outcomes of the pilot-study-II.

8. Industrial Study

As a part of the evaluation process, chapter 8 presents a case study, which investigates the contributions of the thesis. The industrial study based on qualitative analysis investigates the use of VAPs by professional software developers that belong to the UK-based leading software development company. In the end, this chapter

concludes the qualitative analysis results to presents the effectiveness of VAPs in order to raise awareness of vulnerabilities.

9. Discussion

Chapter 9 is the last chapter of the evaluation process, which discusses the conclusive results of the series of experimental studies: pilot study-I, pilot-study-II and industrial study.

10. Conclusion & Future Work

Chapter 10 presents the findings gleaned from developing and applying Vulnerability Anti-Patterns, which details the contributions made by VAPs to increase developers' awareness about vulnerabilities, before critically analysing the thesis in more detail. This critical analysis involves summarising the case made by the thesis, reviewing whether the research questions posed in Section 1.5 have been properly answered, reflecting on the issues identified while conducting this research, and stating how the research contributions in this dissertation answer the research questions and thus validate the overall hypothesis. In the end, this chapter concludes with proposing future work, extending the contributions made.

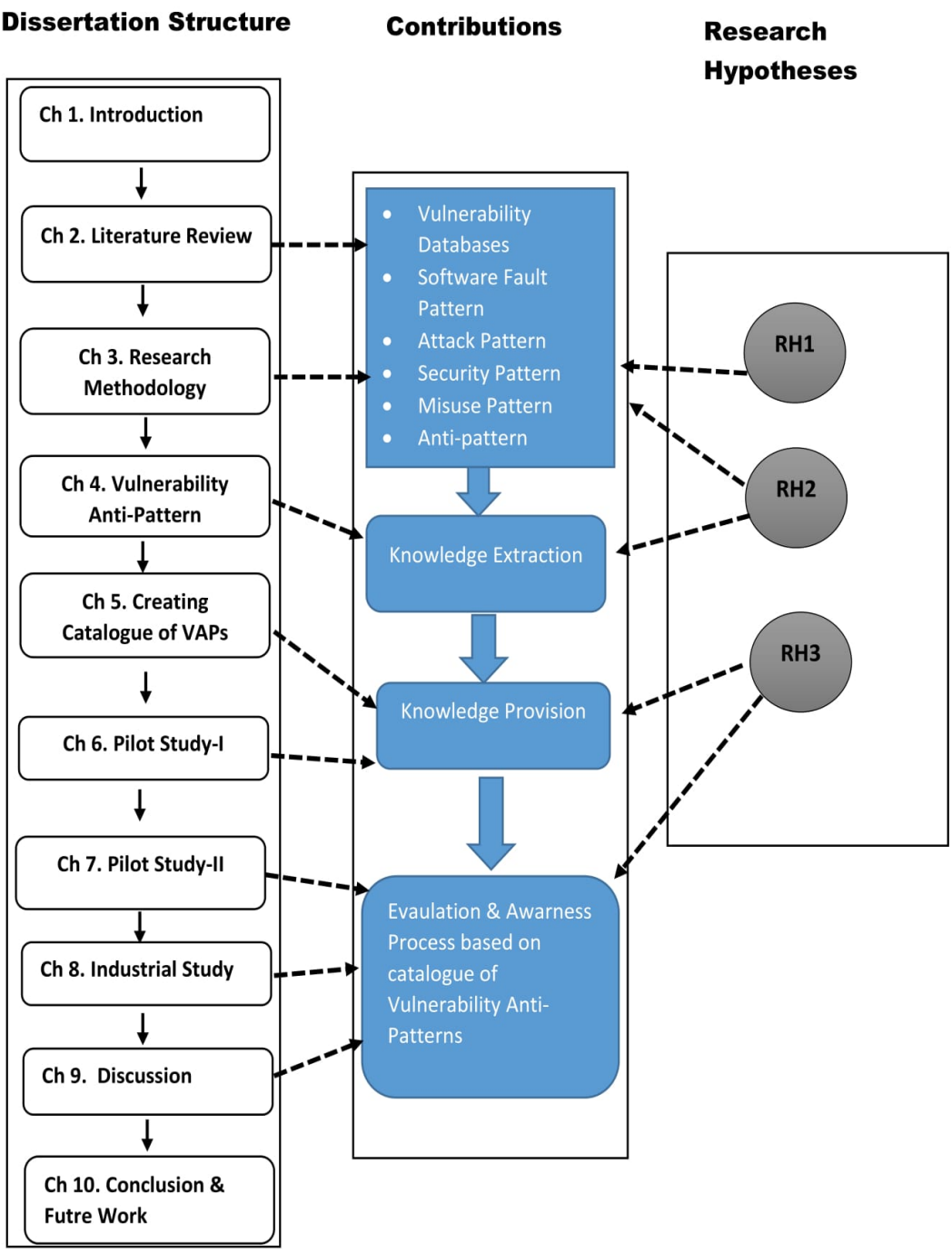


Figure 1 Dissertation roadmap

1.10 Publications Arising from Thesis Work

Table 1 describes the elements of this dissertation which have been published as posters or in conference proceeding and journals.

Publications	Related Chapters
Nafees, T. et al. 2016. Bridging the Void between Software Engineers and Security Experts. In: *Proceedings of the womENcourage 2016 - 3rd ACM-W Europe*, Linz, Austria. 12-13 September 2016. ACM,pp.24-25. **https://womencourage.acm.org/archive/2016/poster_abstracts/womENcourage_2016_paper_24.pdf**	**2**
Nafees, T. et al. 2017. Idea-caution before exploitation: The use of cybersecurity domain knowledge to educate software engineers against software vulnerabilities. In: *Proceedings of International Symposium on Engineering Secure Software and Systems 2017*. Springer, pp.133-142.	**2,3**
Nafees, T. et al. 2017. A Vulnerability Anti-Patterns: Essential Education for Software Developers. *PCWiCS-2017: The Truth About Cyber Security in 7 Words!,* Edinburgh, UK, 10th May 2017. **http://thecyberacademy.org/pcwics-2017/.**	**3,4**
Nafees, T. et al. 2017. Vulnerability anti-patterns: A timeless way to capture poor software practices (vulnerabilities). In: *Proceedings of the 24th Conference on Pattern Languages of Programs 2017*. The Hillside Group, pp.23-47.	**5**

Table 1 Published work from thesis

2 Literature Review

This chapter introduces, some general concepts related to cybersecurity. It is essential to understand, as a part of this research, the extent and characteristics of cybersecurity, while explaining core cybersecurity terminologies from two different perspectives- software developers and cybersecurity experts. In addition, it reviews the literature to answer some critical questions such as: why do we care about security; who is responsible for maintaining security in cyberspace?

Both software engineering and cybersecurity communities share the concept of security in cyberspace; the research evaluates how existing work on vulnerabilities (in the form of vulnerability databases) might be cogent to aid developers to deal with poor software development practices. This chapter provides an overview of efforts made by the software engineering and cybersecurity communities to address security concerns during the software development phases/stages. In particular, potential pattern-based approaches are reviewed, while considering existing issues which arise when transferring security knowledge from cybersecurity community to software engineers when it comes to addressing vulnerabilities in order to create of secure software systems. The discussed core concepts in this chapter will be used in the rest of the dissertation.

2.1 Introduction

Despite significant advances in the state of the art of computer systems in recent years, security of information is lacking, and resources in cyberspace are more vulnerable than ever before. Each major technological advance in software systems raises new security vulnerabilities and threats that require new security solutions. Problems regarding the security of software systems are emerging faster than the rate of the derivation of their solutions. From software developers' points of view, cybersecurity experts developed methodologies that are often too general and hard to understand to implement during software development lifecycle (Kis 2002, Busch, Koch and Wirsing 2014). Software engineering methodologies, on the other hand, lack support to implement security specifications particularly to see errors that cause vulnerabilities (Poller et al. 2017, Xiao, Witschey and Murphy-Hill 2014). Security is considered to protect assets from various threats posed by vulnerabilities inherent in the system. In general, developers lack the particular "security mindset" in order to find

vulnerabilities (Conti and Caroland 2011, Severance 2016). As a result, a distinct communication gap between software engineers and cybersecurity experts exist regarding security (Van and McGraw 2005, McGraw 2012). It is imperative that both communities work together. Questions arising from this are “Why do software developers struggle with persistent software errors?” and “What are the knowledge gaps that exist between software engineers and cybersecurity experts?”

There is no simple answer to these questions, although the most obvious issues are the lack of commonly shared understanding and poor communication between software developers and security experts. In other words, these domains struggle to understand each other’s perspectives and do not share common terms, languages, and procedures, which prevents them from finding causes of vulnerabilities (errors) and explaining them (Fahl et al. 2013). No single security measure or mechanism can provide a completely secure system. In fact, there is a fundamental tension inherent in today’s systems between functionality (i.e. an essential property of any working system) and security (i.e. also critical in many cases) (Hans 2010, Busch, Koch and Wirsing 2014). Therefore, a plan is required to create a balance between systems’ core functionalities and the level of protection required for developing a secure system. We might not be able to change how software systems work in cyberspace, but the transfer of knowledge from cybersecurity community can help in avoiding the typical threats and deploy acceptable security practices (Weir, Rashid and Noble 2017, Nafees et al. 2017).

The start of the chapter presents a history of cybersecurity, which began with computer security and information security early definitions/concepts and how these both merge into cybersecurity. This follows to introduce cybersecurity fundamentals, which are explored from software developers’ and cybersecurity experts’ point of view. This provides an overview of why cybersecurity experts and software developers are different. Following this, cyber-patterns and their relationship within cyberspace are explained. The scope of research was to focus on Cybersecurity and software engineering efforts to deal with software security while using pattern-based approaches. This chapter examined the existing literature related to security from two communities:

- Software engineering community efforts to build security through improved use of software development processes, improving runtime environment and safer programming libraries, and a pattern-based approach (security pattern).

- Cybersecurity community efforts to capture security practice in the form of a catalogue of vulnerabilities and to use pattern-based approaches such as software fault patterns, attack patterns and anti-patterns.

At the end of the chapter, the shortcomings of previous communities' attempts are analysed. The chapter's contents are:

- Section 2.1.1 provides the problem background
- Section 2.2 provides an overview of cybersecurity.
- Section 2.3 presents cybersecurity key terminologies from two different perspectives, i.e. software developers' and cybersecurity experts'.
- Section 2.4 introduces interdisciplinary security related concerts of both communities.
- Sections 2.8 to 2.9 explore the reasons why security is a problem in cyberspace.
- Section 2.10 examines the causes for the distinct knowledge gap between software developers and security experts.
- Section 2.11 discusses the shortcomings of previous pattern-based approaches of both communities, i.e. software engineering and cybersecurity.
- This chapter concludes with a discussion of knowledge pulling and pushing from cybersecurity to software engineering in Sections 2.12 to 2.15.

2.1.1 Background

Programmers make mistakes. Much research effort has concentrated on addressing this problem (Todorov 2015). Of particular concern are those coding flaws that lead to security vulnerabilities. The deliberate misuse of such a vulnerability is termed as exploitation, resulting in information leaks, and reducing the value or usefulness of the system (Leveson 2004). Generally, software developers do not consider the security as their focus is on delivering features, rather than on ensuring the security of the software code, so it is often considered as something to be implemented during later stages of development. However, the cost of fixing bugs post software release is estimated to be 30 times pre-release cost (Cabinet Office 2011). Testing has a poor relation to security. It is unusual for the software developer to use testing approaches for finding vulnerabilities; this issue has not received the research attention it requires (Bekrar et al. 2011). One implication of this is that security concerns should be

embedded into the software development lifecycle (including the early phases) (Jorgensen 2013). In reality, however, software developers struggle against recurring and consistent software flaws (i.e. buffer overflows, and integer overflows), which are exploited by malicious hackers. Nonetheless, a large body of knowledge about software vulnerabilities exists within the cybersecurity community, in particular, amongst penetration testers and ethical hackers. The term 'Ethical Hacker' (EH) will be used as a shorthand to denote this community. Currently, ethical hackers put much effort into classifying discovered vulnerabilities and developing taxonomies of these vulnerabilities. Such vulnerabilities are then catalogued in publicly available vulnerability databases (VDBs) (Aslam, Krsul and Spafford 1996). Software developers have worked to embed security within the software development lifecycle (SDLC) (Howard and Lipner 2006) in order to reduce deployment errors. The mechanism of knowledge transfers between the vulnerability databases (VDBs), developers' perceptions of security issues and the security development lifecycle (SDLC) is complex, which creates a distinct communication gap between ethical hackers and software engineers (Busch, Koch and Wirsing 2014). The application of (knowledge) communication directs software developers to repeat persistent prevalent vulnerabilities and gives rise to software flaws exploitation. Various attempts to capture and formalise the knowledge transfer in a manner appropriate to software engineers have been made, including Misuse Patterns (Fernandez, Yoshioka and Washizaki 2010), Software Fault Patterns (SFP) (Mansourov 2011), and Security Patterns (SP). The need for a better understanding of this mechanism and our proposed solution is the subject of this research.

2.2 Overview of Cybersecurity

Cybersecurity is a field that is continuously evolving, perhaps even more than the IT industry itself (Von and Van 2013). As cybersecurity is undergoing many dramatic changes, it is not easy to explain cybersecurity in a definitive way. In fact, the definition of Cyber or Security are both under debate and the meaning of cybersecurity has evolved over time. For the purpose of this chapter, we will frame the definition in the context of the internet or computer systems' evolutionary history. Thus far, this evolutionary process spans the last five decades. Such evaluation allows us to explore aspects of cybersecurity as explained in Section 2.2.1 and succeeding Section 2.2.3 to define the notion of cybersecurity.

2.2.1 Evolutionary History of Cybersecurity

2.2.1.1 Era of Computer Security

In its origins (Flamm 1988), computer security was focussed on keeping the glass houses in which computer processing units were positioned to protect from vandalism, along with ensuring constant cooling and electricity. There was a limited number of people who used and accessed computer systems (Amoroso 1994, Garfinkel, Spafford and Schwartz 2003). After the revolutionary change in the early 1950's, the internet was designed for researchers to share information easily. In that era, internet as the Advanced Research Projects Agency Network (ARPANET) protocols were introduced by the Department of Defence's Advanced Research Project Agency (ARPA), whose intention was to promote the use of a supercomputer among researchers and share classified information (Abbate 2000). During these years, the primary aim of the internet was to provide openness and flexibility with unrestricted access to information on the network for collaborative research. As a result, the first problems related to information security emerged because of this ongoing design decision unrestricted information sharing. The term information security is loosely defined as the protection of information from unauthorized access, use, disclosure, disruption, modification, or destruction (Fal' 2010), which are explained further in Section 2.2.1.4. Some examples of that time (1980's) security incidents, which raised information security concerns, are described by (Levy 2001) in his book "Hackers: Heroes of the Computer Revolution". Security incidents took place due to unrestricted information sharing over the internet in that era had brought the researchers' attention towards future information security-related issues with a realisation to do work about it.

2.2.1.2 Era of Computer Security or Information Security

As a result of the information sharing and its related security issues, researchers introduced the term *Information Security*, which at first was used synonymously with *Computer Security* (Cherdantseva and Hilton 2013). Regarding the concept of substituting *Computer Security* with *Information Security*, so far there has been little agreement on definitions proposed by Russell and Cherdantseva (Russell and Gangemi 1991, Cherdantseva and Hilton 2013) and (Van and McGraw 2005). These definitions have revealed the relationship between these disciplines - computer

security and information security. There was a general consensus between both disciplines' researchers that the term *Information Security* could be used in an exchange for *Computer Security*. However, according to (Van and McGraw 2005) the definition of *Computer Security* has often been regarded as a branch of *Information Security*, which carefully considers the confidentiality and integrity of classified information, while (Russell and Gangemi 1991) proposed definition embraces all possible aspects of *Information Security* into basic definition of *Computer Security* such as confidentially, integrity, and availability; therefore, the popular conception of *Computer Security* is often called *Information Security*.

During the1960's, which is known as an era of *Information Security*, secrecy and confidentially was the primary security concern across networks and shared computers. By that, security is a matter of protecting the information itself.

The first accounts of malicious hacking and vulnerability exploitation references are: the malicious hacking action published in 1963 in Massachusetts Institute of Technology (MIT's) Tech newspaper (Raymond 2017, Lightsein 2008). The vulnerability exploitation was performed by William D. Mathews from MIT in CTSS running on an IBM 7094 (Csanadi 2015, McMillan 2012). The publicity resulting from the misuse was held to have played a key educational role for users.

In the late 1960's, the networked computer concept was introduced in the form of server and client systems, which enabled the sharing of resources and information, both within a computer and over a network. According to Amoroso (1994), and Garfinkel, Spafford and Schwartz (2003), additional security problems became the primary concern for security researchers and practitioners.

Progress was, however, being made (Cheswick, Bellovin and Rubin 2003) to define *Computer Security* in more detail, so these arising issues could be addressed properly. Consequently, *Information Security* was integrated into computers system security. Authors such as, (Garfinkel, Spafford and Schwartz 2003) propose a more operational definition of *Computer Security* when suggesting that "a computer is secure if you depend on it and its software to behave as you expect", and this concept is often called "trust". In conclusion, however, these informal definitions include natural disasters and faulty software as security concerns. Secure software development and testing concerns in the regard of computer and information security was often overlooked by researchers (McGraw 1999).

2.2.1.3 Digital Era of Cybersecurity

In the early 1980's, the FBI investigated a breach of security at National CSS, as reported in the New York Times (McLellan 1981). This period recognised hackers as being an asset in the computer industry. According to (Walton 2006), the United Kingdom's first Computer Misuse Act was written after a hacking attack on Prince Philip's Prestel mailbox (Calcutt 1999). Such events brought to researchers' attention that the correct operation of software can also become a security problem when it is operated in a manner which it was not intended to be used.

By 1988, the Internet was an essential tool for communications, but it began to create concerns about privacy and security in the digital world. The term digital world here means the virtual space for inter-connected digital devices and media, which is further discussed in Section 2.7.2.

Eisenberg and Gries (1989) described the first computer worm that affected many computers, created by Robert Morris. It was a landmark incident in that it was the first widespread instance of a denial-of-service (DoS) attack and since then this class of vulnerability has been persistently publicised like a buffer overflow vulnerability. Up to that time, developers view that software errors (Kidwell 1998) such as buffer overruns were a potential problem, but not many people realized what the consequences of those errors could be.

The buffer overflow was one of several exploits used by the Morris worm (Orman 2003) to propagate itself over the Internet. Due to the infancy of the internet at the time, the impact was less than it would be today. However, it laid the groundwork for the kinds of security issues that have seem observable since the Morris Worm. This catastrophic incident ultimately forced security practitioners and researchers to include software security as an integral part of information security in cyberspace. Since software errors have the potential to be exploited as vulnerabilities, software security received increased interest among researchers and practitioners (Vacca 2009). In response to the Morris worm incident, the CERTs (Computer Emergency Response Teams) were created in 1988 (Pesante 2002) the first organization of its kind in which researchers coordinate responses to computer security incidents in order to find and publish software bugs that impact software and internet security. The team works with business and government to improve the security of software systems. To deal with obvious and growing dangers of cyber-attacks, a large number of software

exploits emphasised to researchers that bolting security onto an existing system is not a good strategy (McGraw 2012). McGraw argues that software security significantly improved but these efforts are not enough, as he explained in his study in the section on “Bugs per square inch trending down”: Despite the improvement in architecture risk analysis, code review technology and penetration testing techniques, the sheer volume of code product is immense, requiring persistent effort to build secure software. Therefore, McGraw argues the need for training for software development teams to provide knowledge of attacks. He also mentions that exploitation knowledge should be cycled back into the development organization and for developers, thus security practitioners should explicitly track both threat models and attack patterns.

Security is not a simple feature because it requires advanced planning and careful design that includes some emergent properties such as fault tolerance, and error handling. As the internet has greatly increased the connectivity and extensibility of computer systems, and complexity of modern software systems, it is necessary to secure the cyberspace (computers and networks) and its contextual use (information). The question needing to be asked is how developers can implement/adopt the security concerns from the start of software systems development to ensure the use of good software assurance practices. However, improvement required to include software security as an integral part of computer security or information security in order to educate developers about up-to-date software assurance practices. In some way, progress is being made while changing the opinion to security and trying to emerge software security as an integral part of information security (Von and Van 2013). Since the start of the decade, a number of studies have revealed a relationship between information security and cybersecurity that confirmed the association between them in order to protect information (Cherdantseva and Hilton 2013, Von and Van 2013, Alexander and Panguluri 2017).

2.2.1.4 Interrelation between Cybersecurity and Information Security

Following the brief historical evaluation of cybersecurity, this Section will explain how cybersecurity and information security correlate to each other.

According to ISO/IEC 27000:2009, Information security may be defined to preserve three cores of security, namely confidentiality, integrity and availability of information (Fal’ 2010).

According to International Telecommunication Union (Craigen, Diakun-Thibault and Purse 2014), cybersecurity may be defined as “the collection of tools, policies, security concepts, security safeguards, guidelines, risk management approaches, actions, training, best practices, assurance and technologies”, which can be used to protect the cyber environment and both organizational and user assets.

These definitions indicate that cybersecurity and information security are closely interrelated and are, indeed, integrated in order to cover significant aspects of security within cyberspace.

2.2.2 Narrowing the Definition of Computer Security

Before the problem of data (information) security became widely publicised in the media, the idea of computer security mostly focused on the physical machine. Traditionally, computer facilities have been physically protected for three reasons (Stallings et al. 2012):

- To prevent theft of, or damage to, the hardware
- To prevent theft of, or damage to, the information
- To prevent disruption of service

To date, however, increased reliance on the Internet, has radically changed *Computer Security* concerns towards protecting information from unauthorized disclosure, or information secrecy. According to Garfinkel and Spafford’s (2003) definition, *Computer Security,* which was introduced in Section 2.2.1.2, was meant to maintain a “Trust”, which may be defined as a functional component of a system in order to protect and preserve data. Maintaining trust is not an only concern within cyberspace. Referring to ISO/IEC 27000:2009, there are other factors, which need to be considered against sophisticated cyber-attacks. For example, preventing a system from unauthorised access or unauthorised use on the network. These concerns have an association with the core of the concept of *Information Security* such as confidentiality, integrity and availability. Thus far, however, with pervasive remote terminal access now in the form of communication, and networking, the idea of *Computer Security* has been dramatically changed.

According to the NIST handbook’s *Computer Security* definition (Guttman and Roback 1995), it may be defined as the protection afforded to automation systems in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources (includes hardware, firmware,

information/data and telecommunication). From the following NIST definition, which enclosed both *Information Security* and *Computer Security* aspects in much more detail, in addition to the security cores such as confidentiality, integrity and availability (CIA) (Landwehr 1981, Russell and Gangemi 1991) that are discussed further in Section 2.3.2.5, it is suggested that both *Computer Security* and *Information Security* are intertwined in order to maintain security in cyberspace.

It can be concluded from NIST *Computer Security* definition that cyberspace includes security of information/data, network, and internet, which is in line with cyberspace definition introduced in (Blackwell and Zhu 2014b) in Section 2.7.2. After examining the last five-decades of literature in this regard to explore the possible alternative definitions of cybersecurity, the direction of academic thought on the subject of cybersecurity evokes a sense of revolt and revolution, since the security concept was transformed, from computer security to information security, with modern cyberspace ideology. In order to establish a working definition of cybersecurity, the following section presents some proposed formal definitions.

2.2.3 Defining Cybersecurity

In recent decades, cybersecurity has become increasingly challenging, and, in order to comprehend this phenomenon, it is first necessary to clarify the notion of cybersecurity. Different definitions have been proposed, and these have been contrasted with the concepts of cyberspace, information security, and computer security (CNSS, 2003; Butler, 2013; CERT, 2015). Table 2 reflects three different definitions that provide an overview of how cybersecurity and its concept has changed over time.

CNSS (2003)	Butler (2013)	CERT (2015)
Cyberspace	Cyberspace	Cyberspace
	Information security	Information security
	Computer security	Computer security

Table 2 Cybersecurity core definitions in different eras.

As illustrated in Table 2, the most up-to-date definition of cybersecurity comprises three cores: cyberspace, information security and computer security. As shown in Table 2, the definition of cybersecurity evolved over time. Such as, there is consensus to include cyberspace in the definition of cybersecurity, which is

fundamentally considered as a primary core. For example, CNSS (2003) stated, *Cybersecurity* may be defined as the ability to protect or defend the use of cyberspace from cyber-attacks.

CNSS (2003) and Butler & CERT, (2013; 2015) proposed definitions show a modest association between information security and cybersecurity, including the cyberspace.

An advanced definition of cybersecurity is provided by Kaspersky Lab (Butler 2013), describing the practice of defending computers and servers, mobile devices, electronic systems, networks and data from malicious attacks. The term is broader ranging and applies to everything from computer security to disaster recovery and end-user education. To compare the above definitions with Garfinkel and Spafford's (2003) definition, the interesting fact is that computer security is a subset of cybersecurity, because it focuses on protecting computers, networks, programs, and data from unintended or unauthorized access, change, or destruction.

The most comprehensive definition, so far is provided by the CERT Coordination (2015), which defines cybersecurity as "the full range of threat reduction, vulnerability reduction, deterrence, international engagement, incident response, resiliency, and recovery policies and activities, including computer network operation, information assurance, law enforcement, diplomacy, military, and intelligence missions as they relate to the security and stability of the global information and communication infrastructure".

This definition is relatively similar to those of CERT (2015) and Butler (2013) who defined cybersecurity as "the information and communications systems and services composed of all hardware and software that process, store, and communicate information, or any combination of all of these elements: Processing includes the creation, access, modification, and destruction of information. Storage includes paper, magnetic, electronic, and all other media types. Communications include sharing and distribution of information" (Kissel 2013).

Essentially, this reflects that cybersecurity may be loosely defined as computer and network security within the context of cyberspace. With this concept in mind, the term cyber infrastructure has resonated, which requires further explanation in order to articulate a useful concept. Thus, according to NIST provided definition about cyber infrastructure stated that (CNSS 2003): An electronic information and communications systems and services and the information contained therein (CERT 2015, NIST 2011).

There is a consensus in these definitions on having a common security interest in order to define the scope of cybersecurity including computer security and information security, which provides researchers with a means to taking a proactive approach to secure computer systems in cyberspace.

2.2.4 Formal Definition of Cybersecurity

The most acceptable definition of cybersecurity stated that, Cybersecurity may be defined as “the strategy, policy, and standards regarding the security of and operations in cyberspace, and encompass the full range of threat reduction, vulnerability reduction, deterrence, international engagement, incident response, resiliency, and recovery policies and activities, including computer network operations, information assurance, law enforcement, diplomacy, military, and intelligence missions as they relate to the security and stability of the global information and communications infrastructure” (Amoroso 1994, Board 1993, Instruction 2003).

2.3 Fundamentals of Cybersecurity

The study of cybersecurity in this text begins with a description of essential cybersecurity-related vocabulary items and core concepts not inclusive, which can broadly be divided into two sub-categories.

1) **Software Developers**: in the category of developers we include software engineers, programmers, analysts, and testers.
2) **Cybersecurity Experts**: in the category of cybersecurity experts we include penetration testers and ethical hackers.

2.3.1 Core Software Engineering Terminologies

This section describes basic definitions for terms related to software developers, which will be used in this text. Additionally, definitions appear in subsequent chapters to aid in concept understanding. Many of the definitions used in this text are based on the terms described in the *IEEE Standard Collection for Software Anomalies* (Board 1993, Brehmer and Carl 1993, C/S2ESC - Software & Systems Engineering Standards Committee 2010). The standards collection includes the *IEEE Standard Glossary of Software Engineering Terminology*, which is a dictionary devoted to describing software engineering vocabulary (Radatz, Geraci and Katki 1990). It contains working definitions of terms that are in use in both the academic and industrial worlds. All

definitions described in this text have been directly adapted from the *IEEE Standard Glossary of Software Engineering Terminology*, which follows IEEE Standard 610.12-190 (IEEE Standards Coordinating Committee 1990).

2.3.1.1 Defect/ Fault/ Error/ Failure/ Problem

- **Defect:** An imperfection or deficiency in a work product where that work product does not meet its requirements or specification and needs to be either repaired or replaced.
    - **Example:** it includes for example, omission and imperfections found during early life cycle phase (Brehmer and Carl 1993, Board 1993, C/S2ESC - Software & Systems Engineering Standards Committee 2010).
- **Fault:** a manifestation of an error in software. Faults contained in the software are sufficiently mature for detection by test or operation. An incorrect step, process, or data definition.
    - **Example:** For example, an incorrect instruction in the computer program.
- **Error:** A human action that produces an incorrect result. The difference between a computed, observed, or measured value and condition and the true, specified or theoretically correct value or condition.
    - **Example:** For example, a difference of 30 metres between a computed result and the correct result.
- **Failure:** is divided into two main definitions:1) Termination of the ability of a product to perform a required function or its inability to perform within previously specified limits; 2) an event in which a system or system component does not perform a required function within specified limits. Note: A failure may be produced when a fault is encountered. An incorrect result.
    - **Example:** For example, a computed result of 12 when the correct result is 10.
- **Mistake:** A human action that produces an incorrect result.
    - **Example:** For example, an incorrect action on the part of a programmer or operator.

- **Problem:** Difficulty or uncertainty experienced by one or more persons, resulting from an unsatisfactory encounter with a system in use. A negative situation to overcome.
    - **Example:** for example, a login system grant access to unauthorized users.

2.3.1.2 Basic Terminologies Association and Intersection

1. Bug: Fault/ Defect

A bug may be more precisely defined as a fault or defects. As shown in Figure 2, these terminologies are interlinked and generally interdependent on each other. Use of the latter terms related to Bug trivializes that fault has a direct impact on software quality. Use of the term “defect” is also associated with software artefacts such as requirements and design documents. Defects occurring in these artefacts are also caused by errors and are usually detected in the review process (Burnstein 2006). A bug is another name of an error. Software developers usually name it an error; however, testers call it a bug.

2. Usability

According to (Nielsen 1994), usability has multiple components and traditionally may be defined as the ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component. The associated attributes of usability are:

- Learnability: easy to learn
- Efficiency: efficient to use
- Memorability: easy to remember
- Errors: Few and non-catastrophic error
- Satisfaction: pleasant to use

2.3.2 Core Cybersecurity Terminologies

This section describes basic cybersecurity related definitions and terminologies. An additional definition is given in subsequent chapters to aid in concept understanding. Many of the definitions used in this text are directly adapted from *Common Cyber Security Language* (CERT 2015), and *Glossary of Key Information Security Terms* (Kissel 2013), which follows *IEC 27000:2009* standard. It contains working definitions of terms that are in use in both the academic and industrial worlds.

2.3.2.1 Vulnerability

The literal meaning of “vulnerability” is the state of being open to injury. From a computing viewpoint, it may be defined as design or implementation or code errors in information systems or software applications, which may result in compromise of the confidentiality, integrity or availability of information stored upon or transmitted over the affected system (Symantec Corporation 2008). For example, consider Robert’s vulnerability definition (Newman 2009) “a characteristic of a computer system or network that makes it possible for the threat to occur is called vulnerability”. A presence of vulnerability within the software systems provides an opportunity for problems and disasters to occur. Bishop (2003) argues it as opening doors of a system to enable entry to the disallowed state because of a security mechanism failure.

2.3.2.2 Software Vulnerabilities

From a cybersecurity point of view, a vulnerability is a weakness (a form of software error) which allows an attacker to reduce a system’s assurance.

Generally, a vulnerability is the intersection of three features:

1) A system may have a flaw or bug.
2) An attacker may get access to the flaw or bug.
3) An attacker may have the capability to exploit the flaw or bug.

A vulnerability is a subset of a bug. A bug is any defect in a product and different terms are used during the software development process such as a mistake, anomaly, fault, failure, error, exception that discussed in Section 2.3.1. The potential reason to deal with security is to identify the bug before it is exploited by an attacker and is labelled as a vulnerability. Software vulnerabilities share common properties and similar characteristics in general aspects such as location, cause, impact and severity (Chen, Zhang and Chen 2009). Consequently, the process of capturing patterns of vulnerabilities may be effective in order to group and implement standard rules of classification on the enormous population of vulnerabilities.

2.3.2.3 Violations

A violation is a sub-part of vulnerability as shown in Figure 2. It is a malicious or inadvertent action that has the potential to impair the security properties of assets. Violation is sub-divided into four major categories and each category is related to the major class of vulnerability (Schumacher et al. 2013).

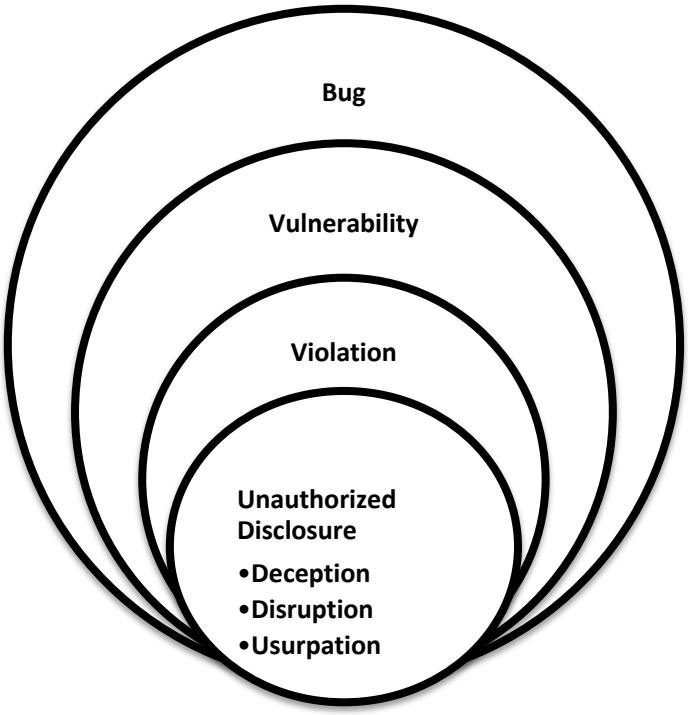


Figure 2 Broader view: class level distribution of software weaknesses

- **Unauthorized Disclosure:** any information that is considered a part of the asset of the organization, which is inappropriately released. There are multiple ways of leakage 1) Exposure: an organization employee leaving a sensitive document on the table at a coffee shop, 2) interception: unauthorized access of data from outside the organization, 3) inference: indirect access by reasoning and 4) intrusion: bypassing the security protection. The purpose is:
    - **Deception:** it includes falsification, repudiation and spoofing. All cases that lead to false information about the organization are part of the deception.
    - **Disruption:** It contains injury or damage to the working interior of the organization. The main sub-parts of disruption are incapacitation, corruption, and obstruction.
    - **Usurpation:** It causes misappropriation theft and misuse of the physical or logical resources of the organization.

2.3.2.4 Attack/ Countermeasure/ Threat/ Incident

- **Attack:** An attempt to gain unauthorized access to system services, resources, or information, or an attempt to compromise system integrity.
    - **Example:** for example, a malicious attacker exploits the web-system with SQL injection.

- **Countermeasure:** Actions, devices, procedures, or techniques that meet or oppose (i.e., counter) a threat, a vulnerability, or an attack by eliminating or preventing it, by minimizing the harm it can cause, or by discovering and reporting it so that corrective action can be taken.
    - **Example:** for example, the developer carefully used the potentially dangerous function calls *strcpy_s* as a safe replacement during coding to reduce the chance of exploitation.
- **Threat:** Any circumstance or event with the potential to adversely impact organizational operation (including mission. Functions, images, or reputation), organizational assets, individuals, other organizations, or the Nation through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of services.
    - **Example:** for example, credential information is protected with email verification to avoid the threat of misuse.
- **Incident:** A violation or imminent threat of violation of computer security policies, acceptable use policies, or standard security practices.
    - **Example:** for example, agent compromised the organization's confidential information via any malicious act or suspicious event.

2.3.2.5 CIA

CIA (confidentiality, integrity, and availability) are three core concepts of cybersecurity. These terminologies are commonly used within the industry but may take on a more particular meaning in the context of cybersecurity. All CIA definitions are sourced from *NIST Glossary of key information Security Terms* (Kissel 2013).

- **Confidentiality:** may be defined as a 3rd party preserving information assets, which are personal to those authorized by the asset owner.
- **Integrity:** may be defined as 'unaltered information assets', which has not been modified or destroyed in an unauthorized manner.
- **Availability:** may be defined as an ability of a system to provide reliable and timely access to information assets by authorized individuals.

The key impact factors relating to users' information are *Authentication*, *Authorization*, and *Nonrepudiation* (Longstaff et al. 1997).

1. Authorization VS CIA

- The process of verification of rights to perform any task is called authorization.

  Loss of Confidentiality: when information is read or copied by someone not authorized to do so, the result is known as loss of confidentiality. For example, for some types of information, confidentiality is a very important attribute such as research data, insurance records, and medical records.
- **Loss of Integrity:** information can be corrupted when it is available on an insecure network. When information is modified in unexpected ways, the result is known as a loss of integrity. This means that unauthorized changes are made to information, whether by human error or intentional tampering.
- **Loss of Availability:** information can be erased or become inaccessible, resulting in loss of availability. This means that people who are authorized to get information cannot get what they need.

2. Authentication VS CIA

The process of verifying the identity of the user or proof of identity is called authentication.

- **Loss of Confidentiality:** Authentication is verification that users are who they claim to be. When secure information is authorized to the unauthorised user such as false identity (identity theft), the result is known as loss of confidentiality.
- **Loss of Integrity:** When unauthenticated users gain access and modified the information. Such as electronic funds transfer, and air traffic control systems.
- **Loss of Availability:** Availability is often the most important attribute in service-oriented business such as airline schedules. When an authorized user cannot get access to specific services this is generally known as Denial-of-service (DoS) attacks.

3. Non-Repudiation VS CIA

The process of assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's intent, so neither can later deny having processed the information.

- **Loss of Confidentiality:** When information is not protected against an individual falsely denying having performed a particular action.

- **Loss of Integrity:** Having no capability to determine where a given individual took a particular action such as creating information, sending a message, and receiving a message.
- **Loss of Availability:** When an authorized user has no capability to approve information because of specific services being unavailable.

2.4 Interdisciplinary Concepts and Approaches

Software security practices are receiving increased interest among researchers and practitioners. The increased interest is a result of an increase in reported security-related vulnerabilities and incidents of security breaches (Daud 2010). This section describes the relationship between the inter-disciplinary field of cybersecurity and software engineering focused on software security.

2.4.1 Software Development Lifecycle (SDLC) and Software Vulnerabilities

The Software Development Lifecycle (SDLC) is generally considered as an idealised model of the process involved in building software. It is often described as a complex process and can be more challenging for the software developer because of its inherent problem that is referred to as complexity includes the program complexity and the design complexity (Conte, Dunsmore and Shen 1986). Failures, faults, and errors can be introduced and slipped into at any stage of SDLC (Beizer 1990, Beizer 2003). Programmers make mistakes. McConnell stated in his book that “Experience suggests that there are 15-50 errors per 1000 lines of delivered code” (McConnell 1993). As the number of lines of code in a software system increases it always results in increased bugs and this is evidenced from developers’ common developing practices during SDLC (Murdico 2007). In the complex and large software application, even 1 error per 1000 lines of code may constitute a large security risk.

For example, each new version of *Microsoft Windows* carries the baggage of its past errors. As Windows has grown, the technical challenge has become increasingly daunting. Several thousand engineers have laboured to build and test *Windows Vista*, a sprawling, complex software construction project with 50 million lines of code, more than 40% larger than *Windows XP* (Lohr and Markoff 2006). This means that if the software developers of *Windows Vista* only made one error per 1000 lines of code, then Vista had 50,000 errors. Exploitation of any one of these errors, bugs and security violations by the attackers is a potential cause of vulnerability in the

software system. Furthermore, these vulnerabilities can also cause other issues such as the loss of information and reduce the value or usefulness of the system (Krsul 1998, Leveson 2004, Cohen 1999).

A vulnerability can be considered to be a type of a bug as described in Section 2.3.1.2. To protect software systems from being exploited or misused by malicious attackers is one of the potential reasons to identify the bug before it is exploited by an attacker and is labelled as a vulnerability. According to (Schumacher et al. 2013) each category of vulnerability is interlinked with the major class of the bug as shown in Figure 2.

The one critical element shared by vulnerabilities is common initiation patterns and similar characteristics such as location, cause, impact and severity (Chen, Zhang and Chen 2009). Normally, vulnerabilities are deliberately introduced or mistakenly slip through the supply chain during the Software Development Lifecycle (SDLC). They come about in delivered software systems due to acquisition processes and by following the poor software development practices. Poor precautions and lack of measurements easily allow vulnerabilities to pass through other levels during the SDLC. It means, the higher the level of software development, the higher the severity and harm impact of these missed vulnerabilities.

2.5 Empirical Strategies in Software Engineering

Empirical strategies in software engineering are defined “to set up formal experiments, study real projects in industry, i.e. performing a case study, and perform surveys through, for example, interviews” (Wohlin et al. 2012 p.120).

According to Sommerville (2010), experimentation is useful in software engineering practices for the provision of understanding in order to identify the relationship between different factors, and variables. Over the last two decades, software engineering research and practices widely have used empirical studies to validate techniques or analyse the results (Anastas 1999).

In order to evaluate practices and tools related to software engineering, empirical studies are essential. For example, to verify and analyse engineering tools and practices as they solve real problems with real practitioners. Therefore, empirical studies based on experimentation are necessary (Sjøberg et al. 2008).

The introduction of empirical strategies in the area of software engineering serves as the basis of the process for experimentation. The fundamental steps in the

process can be utilised for different types of empirical studies. However, the emphasis is on providing help and guidelines to perform experiments in software engineering. Moreover, 'true' experiments with full randomisation, are challenging to perform. In software engineering, there are different types of experimentation depending on the factors and variables of the study. For example, quasi-experiments are frequently used in software engineering. Quasi-experiments are defined as "an experiment in which it, for example, has not been possible to assign participants in the experiments to groups randomly" (Wohlin et al. 2012 p.12). Quasi-experiments are important, and they can gauge valuable outcomes for research based on intervention. There are three different types of strategies based on the purpose of evaluation, whether it be tools, techniques or methods, and based on the conditions of the experimental evaluation.

1) Survey
2) Case study
3) Experiment

The empirical strategies are neither competing nor completely orthogonal. They offer a convenient outcome; however, some studies might be observed as combinations of more than one empirical strategy.

1) Survey

According to Fink (2003 p.36): "a survey is often an investigation performed in retrospect, when, for example, a tool or technique, has already been in use." The key purpose of gathering quantitative or qualitative data are questionnaires or interviews. The survey results are used to derive explanatory and descriptive inferences.

2) Case study

According to Runeson et al (2012 p.114): a case study is "an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified." Case studies are utilised to understand research assignments, activities or projects. Data is accumulated for a specified purpose during the study. Statistical analysis can be performed on the collected data. Normally, case studies are intended to track a particular attribute or to determine a link between different types of attributes. Case studies are considered lower in the level of control, in comparison to experiments.

3) Experiment

Experiment (or controlled experiment) in software engineering is “an empirical enquiry that manipulates one factor or variable of the studied setting.” During experimentation, different treatments are tested on different subjects. However, other variables are kept constant, while measuring the outcome of the independent variables and their effects. There is a difference in human-oriented and technology-oriented experiments. Human-oriented experiments are defined as “humans apply different treatments to objects, while in technology-oriented experiments, different technical treatments are applied to different objects” (Wohlin et al. 2012 p.76). The main goal is to manipulate multiple variables at fixed levels. The manipulation effect is measured, and statistical analysis can be carried out based on this. Generally, quasi-experiments are used, when it is difficult to apply treatments on subjects randomly.

A Quasi-Experiment is “an empirical enquiry similar to an experiment, where the assignment of treatments to subjects cannot be based on randomisation but emerges from the characteristics of the subjects or objects themselves” (Wohlin et al. 2012 p.11). During experimental studies, statistical methods for inferences are performed to show the statistical significance in order to measure which one method is better in comparison with the other (Robson 2002, Sjøberg et al. 2008, Wohlin et al. 2012).

During an experiment, different conditions are applied to differentiate between two conditions, such as an experimental situation and control situation (Wohlin et al. 2012).

2.5.1 Experimental Study based on Intervention

The guidelines state that the process should start from a clear question, in which the population, intervention, difference, outcome, and contextual background have been made explicit (Dittrich, 2002). In an intervention based experimental study, the following aspects need to be taken into account:

I. The population in which the evidence is collected, i.e. which group of people, programs or businesses are of interest?
II. The intervention applied in the empirical study, i.e. which technology, tool or procedure is under study?
III. A comparison is made of the interventions, i.e. how is the control treatment defined? In particular, the ‘placebo’ intervention is critical, as “not using the intervention” is mostly not a valid option in software engineering.

The outcomes of the experiment should be not only be statistically significant but also significant from a practical point of view. For example, it is probably not interesting that an outcome is 10% better in some respect if it is also twice as time-consuming.

The context of the study must be defined, which is an extended view of the population, including whether it is conducted in academia or industry, the industry segment, and the incentives given to the subjects (Höst, Wohlin and Thelin 2005; Petersen and Wohlin 2009). The experimental designs must also be defined. Weir, Rashid and Noble (2017) have explored eight types of intervention to support secure software development, such as incentivisation workshop, threat modelling, component choice, developers training, static analysis, penetration testing, code review and continuous reminder. According to (Such et al. 2016) intervention effectiveness can depend on two aspects: financial cost and team discipline. Financial cost will mostly depend on the cooperate environment. However, team discipline may see reviews of codes to be a matter of development related to security. However, some of the research found that the probability of taking security initiatives during development processes is quite low (Poller et al. 2017).

2.6 Empirical Evaluations of Anti-patterns

Patterns are an approach to abstracting and capturing, for reuse, knowledge about what made a system or paradigm successful. Anti-patterns are derived from design patterns, which capture good practice in software development. Brown (Brown et al., 1998) introduced the idea of anti-patterns as a way of codifying existing bad practice in the software industry; In contrast to a pattern, an anti-pattern “describes a commonly occurring solution to a problem that generates, decidedly, negative consequences”. Anti-Patterns are useful to provide a specific piece of negative advice, as suggested by (Griffiths and Pemberton, 2005). Unfortunately, the efficacy of anti-pattern has yet to be proven. It may be due to a lack of empirical evidence of the practical implementation of anti-patterns. Experimentation is difficult in software engineering (Juristo and Moreno 2013). One problem is the fact that empirical analysis has to be realistic for transfer to industry applications (Sjøberg et al. 2003). However, professionals are expensive, the setting has to be as close as possible to an industry setting, and the tasks must be chosen realistically (Fittkau 2011). It is difficult to answer a question for the use of anti-patterns to transfer knowledge and when it should be

used. However, the software engineering industry is aware of anti-patterns and commonly use them in their practices (Juristo and Moreno 2013).

There is some empirical evidence on the use of anti-patterns in teaching human computer interaction (Kotzé, Renaud and Van Biljon 2008). According to Van Biljon et al (2004), they may be counterproductive teaching tools and a suboptimal knowledge transfer strategy.

According to Cockburn et al (2005) anti-pattern captures poor software development practices with an explanation of why such practices are common and how they lead to a bad solution. The rationale of anti-patterns is to identify recurring flaws/errors and help other people to avoid making the same errors.

The use of anti-patterns for knowledge transfer during software development is not well-known, but there is a growing trend in order to state instructions in negative ways. However, it is argued by (Koenig 1995; Long 2001) that knowledge transfer through anti-patterns might be useful to provide an adequate understanding of the problem. In this situation, it is essential that the anti-patterns are not regarded as blueprints for the problem, preferably that they guide to a refactored solution (Julisch 2013).

The empirical studies carried out by Kotzé et al (2006) and Kotzé, Renaud and Van Biljon (2008) have conducted some initial investigations into human computer interaction context. Results suggest that anti-patterns are studied in order to avoid pitfalls; however, they can actually create pitfalls in knowledge transfer, if not applied appropriately (Kotzé, Renaud and Van Biljon 2008). However, in the cybersecurity context, awareness of negative practices is exposed through teaching (Conti and Caroland 2011). This means that in order to judge whether a specific solution can be successful, it is necessary to describe the context as completely and as accurately as possible (Juristo and Moreno 2013; Petersen and Wohlin 2009). For example, Shull, Seaman and Zelkowltz (2006) and Höst, Wohlin and Thelin (2005) have provided solid results in the use of empirical analysis to improves defect detection techniques, such as the use of anti-patterns to detect the vulnerabilities during software development. The empirical analysis should be used as evidence to prove or disprove the viability of anti-patterns in transferring vulnerability knowledge to software developers.

2.6.1 Use of Anti-Patterns as Intervention

Empirical studies, in software engineering, are carried out to validate the effectiveness of a tool or technique. They are hard to conduct, but are necessary for evaluation of

their practical implementation, in different circumstances. Examples of empirical strategies in software engineering include surveys, case studies and experiments. Interventions are applied in empirical studies to collect evidence of its effectiveness and validate application in a specific context. In software engineering, intervention-based experiments have been carried out to measure the effectiveness of newly designed techniques or technologies. It can be concluded that the implementation of interventions during experimental study depends on the context, such as the segment of industry and incentives for the subject. There are different types of interventions can support secure software development. However, there are some financial and time related constraints. Anti-patterns are useful to describe poor development practices, but it is difficult to answer a question for the use of anti-patterns to transfer knowledge .i.e. When it should be used and how it can be used. The software engineering industry is aware of the anti-pattern technique but does not commonly use them in their practices. Therefore, quasi-experiments can be used as evidence to prove or disprove the efficacy of anti-patterns, in terms of their ability to convey vulnerability knowledge to software developers.

2.7 Cyber-Patterns

2.7.1 The Notion of Cyber-Patterns

Cyberspace is a territory in which humans interact with technologies, processes, data and networks. To make cyberspace secure, patterns have been used to understand, predict and fix security issues in the cyberspace (Jang-Jaccard and Nepal 2014). The inspiration comes from the use of existing design patterns for cybersecurity (Blackwell and Zhu 2014a). Patterns are commonly used in software engineering to find reusable solutions for commonly occurring problems (Walker et al 2014).

However, cyber-patterns remain a poorly-defined term; thus it is challenging to provide a definition. However, any pattern related to security is considered part of cyber-patterns such as, for example, security patterns and attack patterns. Fundamentally, “cyber-pattern” has been proposed to aid researchers in order to capture and find solutions in cyberspace (Blackwell and Zhu 2014b). Thus, it is suggested that patterns would prompt new conceptual and research methodologies for the benefit of cyberspace studies.

In the field of software engineering, a pattern may be defined as a reusable template, which represents a discernible regularity of the design (Gamma et al. 2008). Primarily, software pattern theory is borrowed from Alexander (1977), who described the philosophy of architectural design patterns. This concept takes into account the ability of software engineers to evolve software development processes and to deduce software patterns such as requirement analysis patterns, process patterns, and testing patterns. It is believed that cyber-patterns are more abstract whereas other sciences propose patterns such as in the field of mathematics patterns are definite and always provide definite results. Further, cyber-patterns cannot predict precise results or answers, although output may be observable only by analysis (Blackwell and Zhu 2014a). Generally, cyber-patterns present a "descriptive view" rather than a "perspective view". Cyberspace contains a set of validated patterns, in which each of them describes and predicts one subset of the phenomena. To conclude, cyber-patterns within cyberspace formulate a scientific foundation in order to categorise, classify and organise cyber-patterns through assigning a common language. This is explained further in Section 2.12.2. Vulnerability related cyber-patterns literature was analysed to identify the reasons why existing cyber-patterns are ineffective in providing useful knowledge of vulnerabilities.

- **Design Pattern (DP):** presents a template to build software in an ideal situation (Gamma et al. 2008). The pros and cons of DP to provide vulnerability knowledge are:

Pros	Cons
•Standard format •Easily Understandable to developers •Agreed by Software engineering community	•No association with VBDs •Lack the security knowledge •Insufficient awareness of exploitation circumstances

- **Security Pattern (SP):** describes a security mechanism against potential threats (Brenner 2007). The pros and cons of SP to provide vulnerability knowledge are:

Pros	Cons
•Sufficient security knowledge •Understandable description about the threats to point out the vulnerability	•Lack of standard format •Difficult to understand and implement •No association with VDBs •Inconclusive debates by software engineering community

- **Software Fault Pattern (SFP):** indicates the faulty computation of the vulnerability (Mansourov 2011). The pros and cons of SFP to provide vulnerability knowledge are:

Pros	Cons
•Strong association with VDBs and cybersecurity community •Full explanation of vulnerability (faulty computation)	•Absence of configuration and standardization •Difficult to understand and implement •Complicated structure •poor communication with software developers' community

- **Attack Pattern (AP):** presents a template to describe attacks against software error (Blackwell and Zhu 2014b p.115). The pros and cons of AP to provide vulnerability knowledge are:

Pros	Cons
•Strong association with VDBs and cybersecurity community •Complete explanation of attack •In depth study of exploitation techniques	•Difficult to understand and implement •Complicated structure •poor communication with software developers' community

- **Anti-pattern:** explains a negative mechanism of poor software development common practices (Julisch 2013). The pros and cons of Anti-Pattern to provide vulnerability knowledge are:

Pros	Cons
•Refactor solution for any recurring problems •easily understandable form to assist in software development	•Lack basic structure and format of the implementation •No association with VDBs and cyber security community

2.7.2 Cyberspace

The term cyberspace originated in the early 1980s and since then this notional and augmented environment has been expanding dramatically. There is no agreed definition of what constitutes cyberspace. The earliest definition of cyberspace is “an infinite artificial world where humans navigate in information-based space and as the ultimate computer-human interface” may be defined as cyberspace (Benedikt 1991). Fundamentally, this reflects a virtual internet-based space for users, in which they interact for social, information and creative purposes.

Heim (1991) provides a comprehensive definition that has demonstrated the identified abilities that might be subsumed under the term “Cyberspace”: “is more than a breakthrough in electronic media or in computer interface design, is a metaphysical laboratory, is a tool for examining users’ sense of reality with its virtual environments and simulated worlds”.

Morningstar and Farmer (2003) argue that cyberspace is defined more by the social interactions involved than its technical implementation. However, in recent years, research in cyberspace has grown and evolved significantly (Blackwell and Zhu 2014b). Therefore, the researchers assign a conventional meaning to the term “cyberspace”, which generally refers to the internet and its diverse range of information systems environment for users (Maymí et al. 2018, Li et al. 2017). More importantly, this environment offers various services for its millions of participants, so they can influence and interact with each other world-wide. These interactions include online distance learning facilities for students across the world, use of social media such as Facebook and Twitter, use of online banking and e-commerce such as eBay and Amazon. Undoubtedly, this has had a considerable impact on human daily life and has become an essential aspect of modern society. Nevertheless, cyberspace is highly complex and therefore presents challenges in many aspects such as privacy and security.

According to Lee (2008), security is a growing concern in the cyberspace, due to its inherent complexity. Furthermore, developers face difficulties in enforcing security in this virtual-domain (Wright 2015, Ilyin 2015). As cyberspace strongly suggests an enlarged virtual world of complex infrastructure which has a complicated relationship with the real world. Consequently, this poses grave challenges to researchers, who are involved in management, protection and further development of this space.

2.8 Why Care About Security in Cyberspace?

It is hard to find a facet of modern life that does not involve the internet or use of cyberspace, at least at some level. The virtual environment on the internet where computer systems virtually interact is known as cyberspace. Such as online purchases, debit cards, and automatic bill pay are standard parts of modern life.

Since cyberspace is an open communal area for a wide range of millions of users, it requires judicial, supervisory and legislative bodies to maintain checks and balances in this complicated space (Lee 2008). As cyberspace becomes ever more pervasive in daily life tasks, including: e-commerce, e-socialization, e-banking and e-government, the risk of cybercrime is a growing concern. For example, WannaCry Cyber-Attack[2]. For this reason, many countries have declared their information technology space (cyberspace) as part of their national critical infrastructure. Additionally, these countries have been establishing their own cyber defence authorities. Well-known examples are a National Cyber-Crime Unit (NCCU) (U.K National Crime Agency 2017) and a U.S. Department of Homeland Security (DHS 2017). Despite having these monitoring authorities, unauthorized access, confidential information disclosure and various other forms of cybercrime have been increasing at an alarming rate (Symantec Corporation 2016). The consequences of a successful cyber-attack cover a broad range of possibilities. For example, a minor loss of time in recovering from the problem, a decrease in productivity, a significant loss of money or staff-hours; However, a major loss of credibility

[2] https://en.wikipedia.org/wiki/WannaCry_ransomware_attack

or market opportunity, a business no longer able to compete, legal liability, and loss of life (Wright 2015).

2.9 Why is Security a Problem in Cyberspace?

As of 2016, the internet connected an estimated 6.4 billion IoTs (Internet-of-Things) worldwide, including children toys, kitchen appliances, and pacemakers. As the internet is comprised of loosely connected multiple networks, it fundamentally provides multiple channels for connecting a worldwide collection of devices in cyberspace without regards to national or geographic boundaries or time of the day. As the number of activities of individuals, organizations, and nations being conducted in cyberspace is increasing, the security of those activities is an emerging risk. Since a decade ago, the United States of America has perceived these risks and applied the national strategy for securing cyberspace. As mentioned in previous Section 2.7.2, cyberspace is an augmented virtual space, which often poses challenges of security for software developers and cybersecurity experts. Nonetheless, without having an understanding of cybersecurity, attempts to protect people in cyberspace is a difficult task. Some reasons are mentioned below:

2.9.1 Security is Fundamentally Complex

Since the inception of the internet in 1969, it was designed to provide openness and flexibility, and not intended to be secure. Although such an approach was appropriate at that time, it is not one that lends itself to today's commercial, private, and official use. In the intervening years, usage patterns of the internet have radically changed, leading to acknowledge increased attention of cybersecurity (Walker et al. 2014). The literature described in Section 2.2.1 highlights several issues related to the complexity of cybersecurity.

1. **Complex Relationship or Interconnection between Systems and People**

According to (CERT 2015) cyberspace has competed to a complex interconnection between systems and people, due to complex

interconnection; it creates barriers for software engineers in implementing and maintaining cybersecurity. This complex relationship has drawn researchers' attention towards the fine associations between cyberspace, and the internet with its diverse culture.

Essentially, as argued in the cyberspace definition that real people and systems augmentation has an ability to directly influence and affect each other.

Despite efforts by security practitioners and researchers to secure systems, security has traditionally been a battle of wits: cybersecurity experts try to find vulnerabilities; developers attempt to fix these vulnerabilities (patches). However, malicious hackers successfully manage to exploit the flaws of systems. In fact, the main reasons why software developers cannot be confident in establishing fully secure systems are: cyberspace scale, and its complexity of software systems along with its complicated relations with the real world. Arguably, these complexities have imposed challenges to software developers and industry experts in developing, protecting, and operating software systems securely in cyberspace.

2. Usability is an Impediment for Security

A common perception of users about dealing with security is that it is a nuisance (Morgan, 2016). Generally, software developers perceive security as a painful act (Poller et al. 2017); cybersecurity experts acknowledge that evaluation of security is an as an act of finding software weaknesses rather than assisting developers in fixing them. Common users generally consider security to be the responsibility of someone else such as software developers or cybersecurity experts. These biases are debatable and raise a question about usability (as defined in Section 2.3.1.2) with reference to its relationship with security. Amoroso (1994) argues that determining attacker intent and balancing usability are concerns which present as troublesome impediments to increased security.

3. Technology is Oversold

The most common perceptions in regard to achieving cybersecurity are that: it is irrelevant to real-world problems and can never be accomplished in practice, so any effort is doomed to failure (Arshad et al. 2012).

To achieve cybersecurity, there are general perceptions of bias were overlaid that consider cybersecurity is irrelevant to real world problems and it can never be accomplished in practice, so any effort is doomed to failure. The main reasons for these misconceptions are:

- High Level of Expectation

In general, Users expect more from technology than what it does in reality. For example, cloud computing is considered one of the leading developments in modern computing. The illusion of unlimited resource availability improves the ability of an organization to meet the requirement of a wider user-base. As with any other emerging paradigm, security underpins widespread adoption of Cloud computing. This has been highlighted by the surveys conducted by IDC Exchange where security features as the most challenging aspect in the move towards the adoption of Cloud computing (Arshad et al. 2012)

- Misuse of the Technology

Cloud computing, mobile computing, and the internet of things (IoTs) are ever more dynamic in their use of cyberspace. There is no longer a boundary between the virtual and real world, which raises concerns related to provide and maintain the security of cyberspace. Generally, this digitalization has led to an urgent need to set boundaries between systems and human of these computing paradigms in order to use security in real-world contexts, since this diversity has been causing security flaws that demand an integrated security strategy to control vulnerabilities.

4. Security is an Afterthought

There are millions of software systems around the world connected through the Internet, and many organizations have maintained their own web-space, including university departments, government agencies, corporations, schools, and religions. Furthermore, many individuals have

personal websites or have some sort of reliance on cyberspace, be it through the use of online banking, e-shopping, information gathering/sharing, and entertainment. However, along with the convenience and easy access to information come new risks. Among them are the threats as defined in Section 2.3.2.4 that valuable information will be lost, corrupted, stolen, or misused and that computer systems may be vulnerable and corrupted (see Section 2.3.2.3). If the information is saved electronically and is available on the network, it is more vulnerable than if the same information is printed on paper and locked in a file cabinet.

One of the explanations for this lack of protection that developers tend to regard security as an add-on feature (Mouratidis, Giorgini and Manson 2003, McGraw 1999, McGraw 2012). For them, security is an afterthought at best, and is often neglected. It is reasonably accurate to say that developers do not consider security as critical as other system requirements (McGraw 2006). Furthermore, security, if considered at all, generally comes at the bottom of the list of system requirements.

2.10 Who is Responsible for Security in Cyberspace?

As cyberspace is considered a communal place, so the responsibility to make it secure is dependent on a number of professionals. For example, software engineers who are called software developers and; software defenders who are cybersecurity experts. This section explores the efforts from both communities -software engineering and cybersecurity- to implement and restore the security of software systems

2.10.1 Software Developers: Building Security into the Software Development Process

This is an exciting time to be a software developer. Software systems are becoming more challenging to build (Daud 2010). As software systems are playing an increasingly important role in society, it is necessary to build secure software. There have been numerous attempts to address security concerns as a part of the software development processes. Earlier attempts were targeted at the implementation phase of the Software Development Lifecycle (SDLC) and include those based upon improving

libraries, implementation languages, and language processors. These are typified by the work of (Bourque and Fairley 2014, Sutter and Alexandrescu 2004, Shiralkarand 2009). Approaches based on static and dynamic code analysis have been proposed by providing different guidelines, such as the Microsoft Security Development Lifecycle (SDL), and banned functions (Howard and Lipner 2011). Recently, the software engineering community has been emphasising early exclusion of vulnerabilities by considering security issues at all phases of the SDLC. Typifying such approaches, the Microsoft Security Development Lifecycle (SDL), Security Patterns (SP) and OWASP CLASP development life cycle is considered below.

2.10.1.1 Improving Runtime Environments and Safer Programming Libraries

When personal computers were introduced in the late 1970s, operating systems (OS) were assumed to be used by a single user. As connecting to internet increased cybersecurity threats, it became evident that discovery and disclosure of software vulnerabilities would be a continuing feature of Cyberspace. The software engineering community took multiple steps in order to deal with new security challenges.

As reflected in the literature (Howard and Lipner 2009), many security breaches occur due to insecure runtime environments. For example, discoveries of vulnerabilities in *Netscape* and *Internet Explorer* received wide publicity (Allen et al. 2001). *SQL server 2000* initially released in 1999 was issued with 16 vulnerabilities (Howard and Lipner 2009). Given the criticality of securing operation systems, there are many papers that study the distribution of bugs and vulnerabilities due to the adoption of inadequate software practices (Garcia et al. 2014, Miller et al. 1995).

As explained earlier, security threats in cyberspace have changed over time. It is argued that from operating systems in vulnerability data sourced from the NIST (NIST 2011) that software system architects accept could that systems will contain some faults taking a ’less is more’ attitude when cyber-attacks occur. However, patches for fixing vulnerabilities are

generally superficial solutions and often do not get at the core of the software problems (Morgan 2016).

Developers who are continuously challenged in order to build more secure software also trying to improve programming languages by steps such as introducing new more secure runtime libraries, as summarised in Table 3. As a result, developers can more easily avoid those types of dangerous errors that caused vulnerabilities. However, the success of these proposed solutions is doubtful due to many reasons and are discussed further in Section 2.14. This section gives discussion to some secure software engineering practices to reduce vulnerabilities rather than trying to do exhaustive research.

	C/C++	PHP
Safer Libraries	• The security development lifecycle • Secure Coding in C and C • Guidelines for Secure Coding • The design and evolution of C • C coding standards: 101 rules, guidelines, and best practices • Security Development Lifecycle (SDL) banned function calls (Howard and Lipner 2011, Meyers 2005, Seacord 2006, Sutter 2002, Schumacher et al. 2013, Stroustrup 1994)	• Programming PHP with security in mind • Programming PHP with security in mind (Loureiro 2002, The PHP Group 2017)

Table 3 Literature summary in order to improve programming languages with safer libraries

1. C/C++ Banned Function Calls

Over three decades ago, when the C runtime library (i.e. CRT Microsoft terminology) was first created, computer threats were different. Computers were not as interconnected as they are today, and attacks were not as prevalent. Therefore, a subset of the C runtime library has been deprecated

for new code and, over time, removed from the earlier code. Usage of these outdated functions easily leads to vulnerabilities because of its unsecured behaves. Even the provided replacement functions had suggested with no guarantees of security. Because of this danger in C/C++, (Howard and Lipner 2011) a list was compiled of dangerous functions composed of known bad functions that must be avoided to reduce vulnerabilities and need to be updated as part of secure software practices. According to (Howard 2005), It is recommended from their experiences with real-world security bugs (occurred due to banned function calls), which are focused primarily on functions that can lead to vulnerabilities. Existing code must either replace the banned function with a more secure version or be re-architected so that the banned function is not used. Importantly, for the functions marked as “recommended”, developers consider these functions a strong recommendation and evaluate the function against the systems own security requirements, elevating them to “required” as necessary. It is strongly suggested that, none of the listed banned functions should be used in new code during the Software Development Lifecycle (SDLC). For example, developers use dangerous functions such as *strcpy(), strcat(),* and *sprintf()* that do not check bounds and lack the assurance that the bounds will never get exceeded.

Once the program executes, it invokes a potentially dangerous function that would provide an opportunity for a malicious attacker to exploit this vulnerability. However, these functions can be used safely. Microsoft Safe CRT is included starting with Visual Studio 2005 that provides somehow tool support against these dangerous function calls. For example, the Visual Studio 2005 (and later) compiler has built-in deprecations against dangerous functions, developers must investigate all *C4996 compiler warnings* to make sure that the function in question is not on the preceding banned list.

2. PHP Deprecated Functions

A similar approach can be found in PHP programming practices. To address growing concerns related to security, researchers are keeping revising PHP libraries or functions to assist developers in creating more

secure web-applications. For example, PHP 5.3.0 introduces two new error levels: E_DEPRECATED and E_USER_DEPRECATED.

The E_DEPRECATED error level is used to indicate that a function or feature has been deprecated (The PHP Group 2017). They also competed to industrialise their knowledge such as secure PHP programming practices in order to achieve the security posture of web-applications (Loureiro 2002).

3. Microsoft SDL

The software industry's past is littered with security exploitation. Microsoft has learned from and has been proposed the Microsoft Security Development Lifecycle (SDL) to remedy past mistakes. The Microsoft SDL may be defined as Microsoft's security assurance process, which builds security into every phase of software development and offers defence-in-depth guidance and protection. It suggests a practical and holistic approach to addressing security concerns of developing software and to implement security enforcing postures such as privacy and reliability. Furthermore, incorporation of Cybersecurity standards, such as *ISO 27001* [30] into Microsoft SDL ensures that any software produced with this process complies with industry recognised standards. The Microsoft SDL goes beyond the traditional software development process and incorporates the overarching information security management system that requires the specification of security guidelines for policies, processes, and systems within an organization. The Microsoft SDL is continuously evolving and improving to deal with ever more complicated, and sophisticated attacks (Howard and Lipner 2006, Howard and Lipner 2009). In contrast to traditional SDLC, Microsoft SDL is an add-on with two additional phases. For example, a core security training phase and repose phase (Howard 2005). Usability analysis of Microsoft SDL is discussed further in Section 2.14.2

4. OWASP CLASP

OWASP CLASP - short for Comprehensive, Lightweight Application Security Process - has been developed to embed security considerations

during the early stages of the Software Development Lifecycle (SDLC) for web-applications. It includes a set of guidelines for web security requirements, cheat sheets, a development guide, a code review and a testing guide, an Application Security Verification Standard (ASVS), a risk rating methodology, tools and a top 10 of web security vulnerabilities. This is explored further in Section 2.14.2.

5. SP

Security Patterns (SP) are used to describe a solution to stop or mitigate a set of specified threats through certain security mechanisms. These are patterns designed to assist software developers who are not security experts with embedding security in their systems. It can also be a useful tool for teaching security concepts (Brenner 2007). This is explored further in Section 2.13. Security Patterns are an enhanced form of design pattern for software developers' assistance to add security inside their applications. They are not, however, based directly on the vulnerability knowledge stored in VDBs which is necessary for achieving currency and timely response to new threats (Halkidis, Chatzigeorgiou and Stephanides 2006).

2.10.2 Cybersecurity Experts: Attempts to Capture Security

Approaches to the problem of building security domain knowledge into the Software Development Lifecycle originating from the cybersecurity domain can be considered under two headings: Attempts to catalogue vulnerabilities and attempts to communicate vulnerabilities using patterns.

2.10.2.1 Attempts to Catalogue Vulnerabilities

The National Vulnerability Database (NVD) comprises CWE, CVE, and CAPEC which are the three most comprehensive vulnerability databases (VDBs). They are open-source and maintained by MITRE (MITRE Corporation 2013, MITRE Corporation 2015b, MITRE Corporation 2015a), based on the Security Content Automation Protocol (SCAP).

- **CWE:** The Common Weakness Enumeration database (CWE) catalogues weaknesses that can occur in software. These

weaknesses are described as software bugs that can lead to vulnerabilities. For example, in Figure 3, CWE- 250: Execution with Unnecessary Privileges, and as such can be considered as an abstract, top-down view of the types of errors that can occur in software. The CWE aims to raise awareness and understanding of software flaws in software in order to eliminate these from released versions.

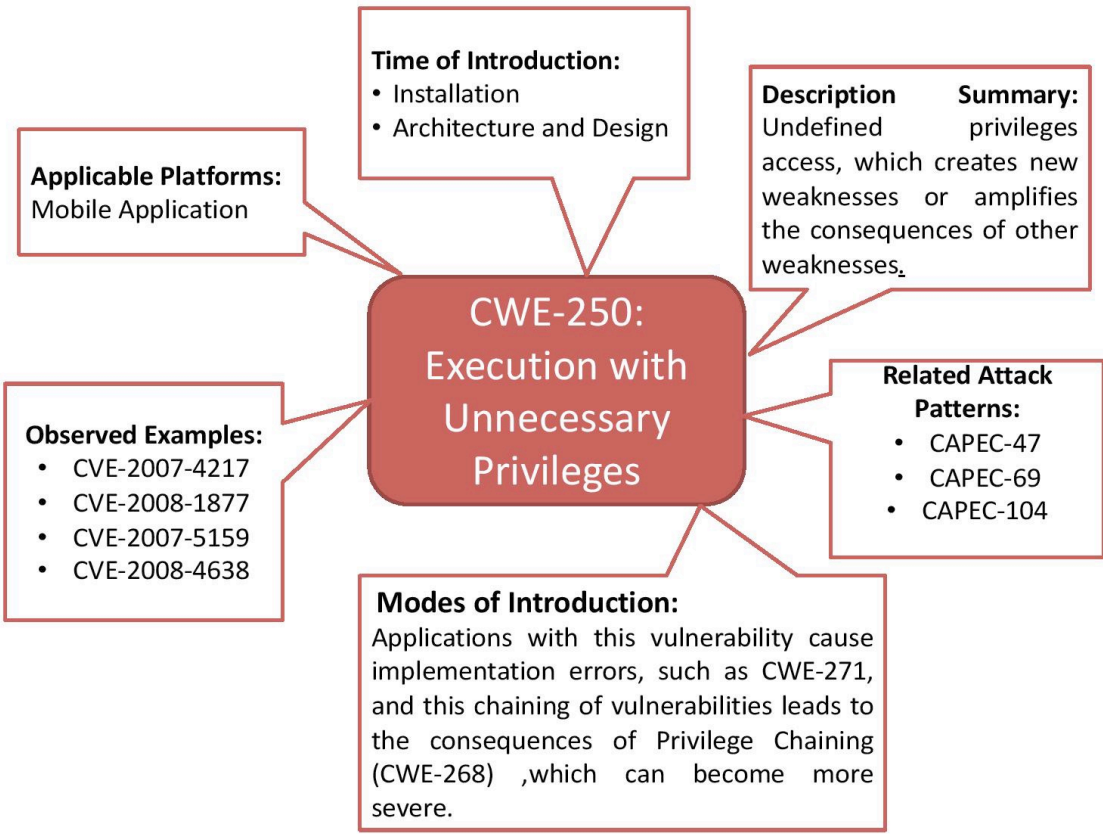


Figure 3 Vulnerability information in CWE-250

- **CVE:** The Common Vulnerabilities Enumeration database (CVE) catalogues specific examples of publicly known vulnerabilities that exist in software and is designed to facilitate the sharing of information about these vulnerabilities across a number of different capabilities, including IDS, scanners, repositories. For example, CVE-2007-3931. A malicious attacker can execute arbitrary code and successful exploitation may result in compromising rights of the system.

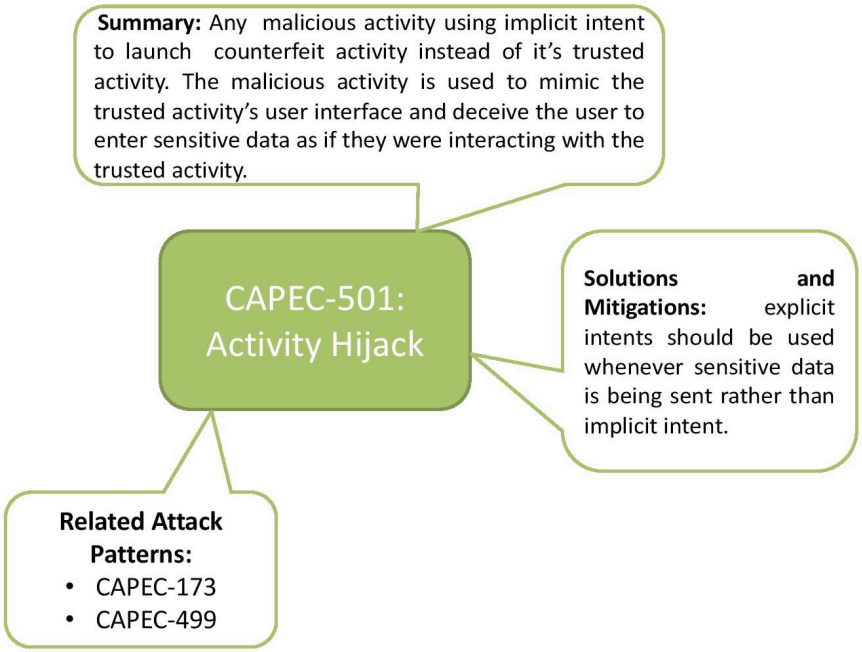


Figure 4 Attack pattern example in CAPEC-501

- **CAPEC:** The Common Attack Pattern Enumeration and Classification database (CAPEC) provides formal attack patterns and is designed to bridge the knowledge between the CWE and the CVE and offer guidance to software developers on how software weaknesses are likely to be exploited by a malicious hacker as shown in Figure 4, which demonstrates attack pattern example: CAPEC-501.

All of the above described databases are organized by NVD (NIST 2015), which enables these databases to contain information to be interlinked and searchable as shown in Figure 5. To improve usability and functionality, these databases share similar formats, styles and fields. There are, however, problems surrounding information redundancy, information conflicts and information representation across these databases such as methods of attack, CIA impact and mitigation techniques (MITRE Corporation 2015b). As such, it can be especially challenging for software developers to implement security recommendations. These databases will be critiqued further in the succeeding Section 2.15.

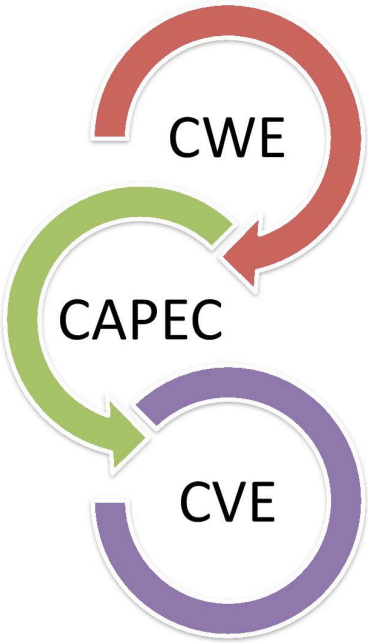


Figure 5 Interconnection of vulnerability databases (VDBs)

2.10.2.2 Attempts to Use Patterns to Communicate Vulnerabilities

In addition to the above VDBs, security experts have also attempted to embed vulnerability related security knowledge in the form of patterns such as SFP, AP and Misuse pattern. Figure 6 presents the author's proposed conceptual VDBs information flow model, which explains how to bridge the gap between VDBs developers and users. This will be explored further in Section 2.13.

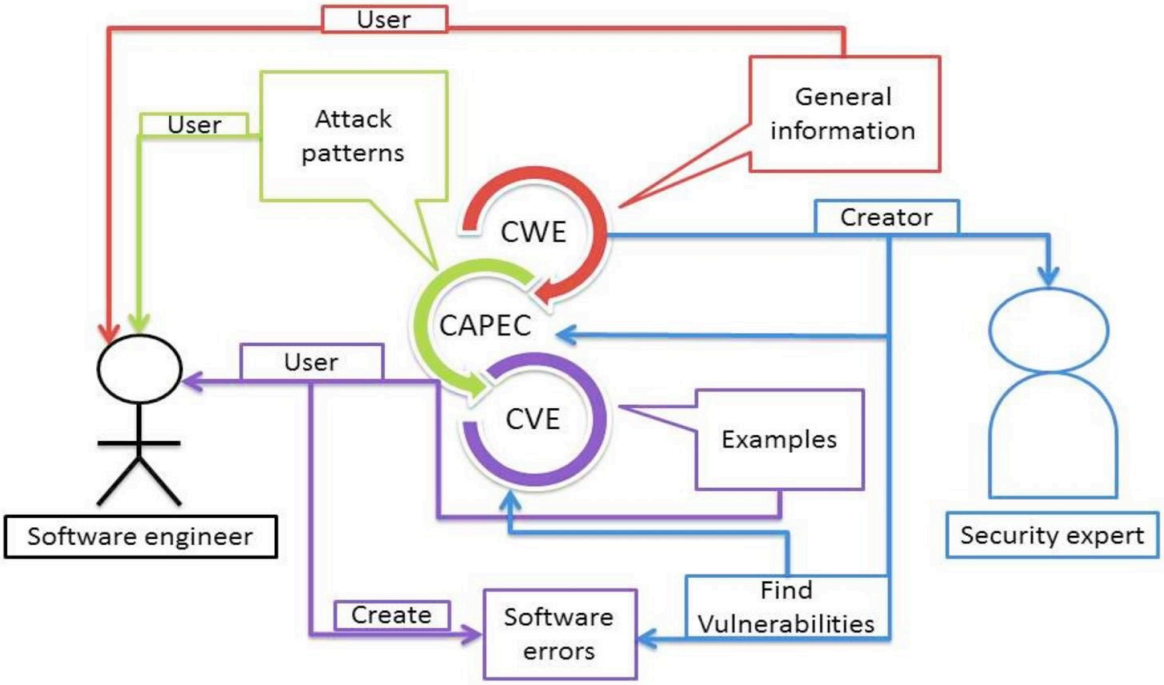


Figure 6 Vulnerability information flow among VDBs

1. Software Fault Pattern (SFP)

SFP (Mansourov 2011) has been designed to provide a formal specification of weaknesses (vulnerabilities) and are aligned with the CWE database. These will be explored further in Section 2.13. The idea of SFPs is based upon intelligence from CWE that might help to make such information more understandable and easier to use. However, a lack of detailed information about the structure and format of SFP presents a considerable obstacle for software developers (Arnold, Hyla and Rowe 2006).

2. Attack Patterns (AP)

AP sources information from the CAPEC database, which describes a procedure of a particular vulnerability attack format. It is not intended as a source of design patterns (like standard software pattern) and software developers' attitude towards APs is not an effective means of understanding in regard to vulnerability attack. Software developers do not use AP to find the vulnerabilities due to their lack of understanding (Bunke,

Koschke and Sohr 2012). In other words, they fail to appreciate the purpose of the AP fundamental to support developers in order to provide an understanding of attacks and its procedures. This research has shown that APs are rarely used by software developers' due to their inherent complexity, but also because AP, tend to be written in a format that is not easily understood by software developers (see Section 2.13).

3. Misuse Pattern

A misuse pattern describes the malicious hacker point of view in a generic way while considering sub-dimensions to classify it as a set of attack actions and enumerating with possible security patterns as a countermeasure (Schumacher et al. 2013). Although, the misuse pattern groundwork clearly illustrates that VDBs sources are not utilised in defining its attack actions, there is definitely mutual knowledge sharing with security patterns. Furthermore, so far there are not well-supported pieces of evidence of their practical usage. This clearly has shown that misuse patterns have certain construction deficiencies and lack considerable usage for software developers (see Section 2.13), such as undefined attack pattern knowledge sources and less practical values.

4. Anti-Pattern

Anti-pattern "describes a commonly occurring solution to a problem that generates decidedly negative consequences". While there are no solutions against prevailing vulnerabilities, it is practicable to address recurring and frequent prevalent software vulnerabilities through anti-patterns (Julisch 2013). From this, the question that is formed is "which pattern will be considered for better transformation into an anti-pattern particularly as a solution?" One of the aims of this research is to determine which mechanism of integration will lead to enabling the different patterns' (such as SFP, AP) transformation in generating a general solution (anti-pattern). It appears one of the most appropriate and communicable solutions for providing awareness of vulnerabilities and an effective understanding to software developers (Walker et al. 2014).

2.11 Why There Exists a Distinct Knowledge Gap Between Software Developers and Cybersecurity Experts?

One of the essential elements shared by every modern information system is the software that determines how the system behaves. Today's software problems have led to spectacular real-world failures of many different kinds, including security problems, reliability problems, and safety problems. It is probably only a matter of time before software causes the demise of a large company. What do software developers need to do in order to combat dangerous software errors occurring especially due to the rush to embrace e-commerce and the intense pressure of Internet time?

2.11.1 Inadequate Knowledge Sharing

The lack of a shared understanding of both communities - cybersecurity and software engineering - is well-documented (Howard 2004, Borstad 2008, Bunke, Koschke and Sohr 2012, Arnold, Hyla and Rowe 2006). This gap in communication generates biased knowledge partly by the way the practices of penetration-testing/ethical hacking has evolved, and partly due to cultural/sociological factors between the two communities. Although there are exceptions, security testing sometimes takes place as an activity separate from the SDLC. Cybersecurity experts such as penetration-testers communicate with and report to system administrators and IT managers. There is relatively little communication from those doing security testing to that building software. There is no (formal, methodological) feedback path from the security testing activities to software development activities (Poller et al. 2017). Lack of knowledge is present in individual software engineers, software engineering teams and organizations as a consequence of being unable to learn from poor software practices (vulnerabilities knowledge from VDBs). The same kinds of insecurities are built into the next generation of software systems.

This lack of awareness was also in evidence due to some prejudiced cultural reasons among these communities. Although, there is some crossover between the basic knowledge and skill-set of a software engineer and an ethical hacker/ penetration-tester, they represent some

very distinct technical domains, with different educational paths, different technical languages and different professional bodies. To fill the basic knowledge gap the teaching of cybersecurity during undergraduate Computer Science courses can help (Roumani et al., 2012; Fahl et al., 2013), and reflects improvement evidenced through the practices of recently employed software engineer graduates, those likely to have a better appreciation of techniques of support) cybersecurity than one whose education is less recent, software engineering (SE) teams are likely to include a variety of ages/experiences.

Software developers are responsible for the development of software systems that will represent their company's reputation and ensure the safe conduct of business online, thus, appreciate the seriousness of creating secure software. Despite this, discussions around security are often avoided because instigating meaningful change is too complex, too slow and too expensive. There is a general perception that development mistakes are caused by inexperienced developers or developers who do not understand the tool, language or technology; however, this is not the case. In fact, even experienced developers make common security mistakes. The more senior engineers/managers may give a lower priority to producing secure code than to producing working code. Mistakes made by software developers are generally seen as the primary cause of security flaws in software systems. We argue instead that the fault lies with the process: developers lack an understanding of how malicious hackers can exploit software flaws, and this understanding is necessary for the creation of secure software. One key explanatory factor for this is a lack of awareness about poor software practices that cause exploitation.

One of the best solutions proposed by (Yun-hua and Pei 2010) is the creation of an 'ecosystem', bridging the knowledge gap and providing a common ground of understanding to ethical hackers/ penetration testers and software developers to facilitate in collection and accumulation of cybersecurity knowledge needed for security assurance, and to ensure its efficient and affordable delivery to software developers (Acar et al. 2016) to the defenders of cyber-systems, as well as to other stakeholders.

2.11.2 The Hacker's Time Advantage

Generally, a malicious hacker does not work under the same constraints of project schedules and deadlines as a software engineer does. If they wish to spend six months examining, in minute detail, the state of the stack under a particular attack condition, they will not have employers pressurising them, to deliver. Thus, they have the advantage of time. This, coupled with the extensive knowledge-sharing that takes place amongst the hacking community means that a hacker may be more familiar with the weaknesses of a particular piece of software than those who created it, which is explained in Section 2.9.1. The evidence for this is the frequency of newly discovered vulnerabilities. Ethical hackers and software developers, on the other hand, can lack efficiency in their knowledge sharing due to corporate barriers (Van and McGraw 2005, McGraw 2012).

This research concluded that the main reason for software developers' lack of security understanding is because their focus is on delivering features rather than on ensuring security. Accordingly, developers often consider security as something to be added to a system as a bolt-on component in later stages of development (McGraw 2012).

2.11.3 Lack of Knowledge Industrialization

As software systems become ever more complex and connected by the internet, security is a growing concern. The frequency and reoccurrence of mostly discovered vulnerabilities undoubtedly confirm that poor software practices are continuously adopted, repeated, and implemented by developers during the Software Development Lifecycle (SDLC). It may be fruitful to study failures and identify recurring problems such as poor software practices to find solutions against these problems. Some researchers (Mansourov and Campara 2010, Shiralkarand 2009) have mainly been interested in a question concerning: why software developers generally overlooked security issues throughout the Software Development Lifecycle? In particular, as mentioned vulnerabilities or faults (Section 2.3.1.1) are introduced accidentally through the supply chain and slip through into delivered systems due to the adoption of poor software

practice. The industry has realised that with traditional system security engineering, the error-free, the failure-free, and the risk-free operation is not usually achievable, within acceptable cost and time constraint during SDLC. As far as security is concerned in the software development process, there is no single technique, which could be considered as a 100% secure. Arguably, software developers have consistently failed to develop secure software systems (Ilyin 2015). One question that needs to be asked, however, is whether during the software development process, developers intentionally make the development mistakes or whether they lack the necessary understanding of 'how a malicious attacker can exploit/misuse these software development errors'. Arguably, malicious hackers have a time advantage, indicating that the effectiveness of attackers can be traced back to their extensive knowledge sharing (see Section 2.11.2.) However, security experts and software developers in this regard, fail to share their knowledge with each other efficiently, and although the problem of frequently recurring software vulnerabilities is very well known, no standard solution has been universally adopted (Aslam, Krsul and Spafford 1996). Questions have been raised, such as how both communities' experts capture and share their experiences of poor software practices (failures in solving problems) in a form that is suitable for the other party that is with clarity, rationale, and context in a way which could be applied to a new solution.

In theory, all software systems have some vulnerabilities as discussed in previous section 2.3.2.2; whether or not they are serious, depends on whether they are used to cause damage to the system. In cyberspace, poor software development may be considered as one of the most serious threats, and the principal reason for this is a lack of information sharing about the prevalent software flaws that can easily lead to vulnerabilities (Morgan 2016, Busch, Koch and Wirsing 2014). According to McGraw (2012) the existing software design and engineering processes provide little guidance about preventing security exploitation during SDLC. This information disconnects (gaps) between software developers and cybersecurity experts have led to widespread software vulnerabilities

(Kalaimannan and Gupta 2017). The researcher proposed an idea called “Caution before exploitation”, which basically means to provide essential security awareness to developers through industrialised security knowledge. To be successful, these knowledge forms need to be presented in an understandable format (Acar et al. 2016, Fahl et al. 2013). This would consequently lead to an efficient “safe environment” which could amplify in order to industrialise the necessary security knowledge from a few highly skilled developers/security experts. This would transfer necessary security knowledge to a large number of less skilled but highly motivated developers.

2.12 What is Pattern-Oriented Research Methodology?

“Cyber-patterns are predictable regularities in cyberspace that may help us to understand, design, implement and operate successful and reliable cyber systems.” (Blackwell and Zhu 2014b). The pattern-oriented methodology has been proposed by (Blackwell and Zhu 2014a), to introduce the concept of reusability of cyber-patterns. For example, the methodology applied to the design pattern to answer the following research questions

- How to identify and document design patterns
- How to catalogue and develop pattern language of design patterns?
- How to design formal specification of design patterns?
- How to develop software tools from design patterns?

2.12.1 Why Use Patterns to Find the Solution?

The pattern-oriented research methodology is used to address problems through the general features of patterns:

- Finding the regularities of complicated problems in cyberspace.
- Providing the core understanding of reoccurring problems through the discovery of underlying mathematical structures.
- Facilitating the observation of problems in order to detect the occurrences of its repetition.
- Defining the operations to predicts, detect and prevent problems.

- Devising the mechanisms of solutions for classifying and categorising problems.
- Formulating the standard set of vocabulary of the recurring problems to form pattern languages.

2.12.2 How Do Cyber-Patterns Interact and Interrelate with each other?

As stated previously in pattern-oriented research methodology (Section 2.12), cyber-patterns are certainly interlinked and connected to each other. The interaction of similar types of cyber-patterns is known as composition (Riehle 1997), hybridization is defined as an interaction of different types of cyber-patterns.

Furthermore, these types of interaction among cyber-patterns significantly provide a more effective means to use and protect the infrastructure and resources on the internet. For example, a security pattern is sub-set of design patterns and each security pattern protects or detects more than one type of attack patterns.

2.12.3 Cyber-Patterns for Vulnerabilities

Evaluation of cyberspace that includes progressively complex infrastructure, is a complex task specially to find solutions via applying the contemporary approaches. The Cybersecurity community has made efforts to identify, categorise and classify software vulnerabilities. However, such information lacks formal representation in the form of patterns (McGraw 2004). Besides, cyber-patterns individually (without having the cybersecurity domain knowledge) are insufficient to provide an understanding of prevalent vulnerabilities. Consequently, software errors are constantly reoccurring, and potential reasons for this are: lack of intuitive knowledge of vulnerability databases, outdated knowledge sources, and insufficient awareness about underlining root-causes (Busch, Koch, and Wirsing 2014, Ghani et al. 2013, Yun-hua and Pei 2010).

Generally, cyber-patterns' core objectives are to support software developers while providing guidance and assistance during the software development process (SDLC) different aspects. For example, design

patterns consider the functional or non-functional requirements aspect of the system, security patterns concern security issues of the system and attack patterns focus the security attack procedures of the system. Nevertheless, most of the cyber-patterns do not have sufficient evidence of their industrial use and lack detailed descriptions of their practical implementation during software development. The main concern is whether these cyber-patterns themselves give information about prevalent software flaws that can easily be exploited by the attacker as a vulnerability. This means that the education of software developers is paramount to successes on those patterns that are related to security flaws of software weaknesses (vulnerabilities), rather than the existing design patterns that only focus on functional and non-functional requirements.

The research performs a comparative analysis of relevant cyber-patterns, which have been carefully selected to fulfil developers' need in order to attain adequate information about vulnerabilities. Ultimately this will determine the most effective way of repeating vulnerabilities information. Security knowledge intervention will increase the software developer's ability to identify and mitigate software vulnerabilities. Thus, there is a definite need to configure/channel cybersecurity community efforts related to vulnerabilities into a formal organisation. This formalisation could certainly help software engineers to understand vulnerabilities root-causes through assigning a common language, namely a pattern.

The work in this thesis proposes a solution based on *hybridisation* of patterns which allows the generation of an optimal solution, referred to as a "*Vulnerability Anti-Pattern"*. As shown in Figure 7 design patterns, security patterns, and attack patterns have been selected from the list of cyber-patterns.

2.13 The Shortcomings of Previous Pattern-Based Approaches are Sourced from Both Communities - Software Engineering and Cybersecurity

Trying to capture and address security in the form of patterns represents an on-going issue amongst cybersecurity experts and software engineers. For example, Figure 7 presents the existing cyber-patterns, software engineers attempt to capture security (see Section 2.10.2.2), which is described as Security Pattern, Misuse Pattern, and Anti-Pattern; cybersecurity experts' efforts to use patterns to communicate vulnerabilities Software Fault Pattern and Attack Pattern.

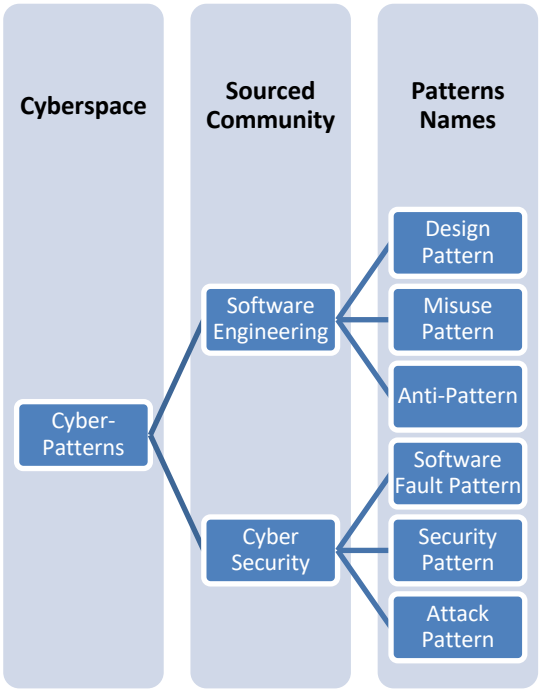


Figure 7 Existing cyber-patterns capture security in cyberspace

1. SP (Security Pattern)

As discussed in Section 2.10.1, the SP defines the security forces and its consequences, which are summarized in Figure 8. However, there exists little research on the categorization of SPs, which have been proposed for use against vulnerabilities. It is still not clear to inexperienced developers what pattern should be adapted and applied during development. Other impediments that are often present to developers include the lack of communication with cybersecurity experts' knowledge as sourced from VDBs having considerable significance to achieve currency and a timely

response against new threats (Halkidis, Chatzigeorgiou and Stephanides 2006).

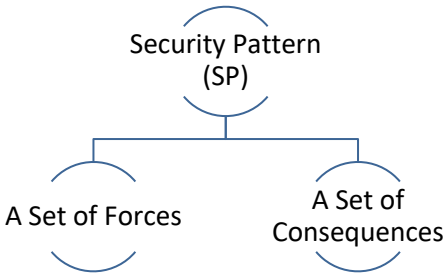


Figure 8 Anatomy of security pattern

In addition, it is evidenced by the literature exemplified in Table 4 that there is a number of proposed catalogues to organize security patterns. For example, *An inventory of security patterns* (Yskout et al. 2006), *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management* (Steel and Nagappan 2006), *Classifying security patterns* (Fernandez et al. 2008), *A Natural Classification Scheme for Software Security Patterns* (Alvi and Zulkernine 2011), *Enterprise security pattern: a new type of security pattern* (Moral-García et al. 2014), and *Vulnerability-Based Security Pattern Categorization in Search of Missing Patterns* (Anand, Ryoo and Kazman 2014). Despite all of these above efforts, it does not work due to the lack of standard format and lack of association with VDBs.

Key Issues	References
• **Lack of standard format** • **No association with VDBs** • **Inconclusive debates by software engineering community vulnerability**	(Bunke 2015, Yoshioka, Washizaki and Maruyama 2008, Skout, Scandariato, and Joosen 2012, Borstad 2008),

Table 4 Security patterns key issues

2. Misuse Pattern

As discussed above in Section 2.10.2.2, the Misuse Pattern describes the attack, forensic data and mitigation, which are summarised in Figure 9.

Although, the misuse pattern groundwork clearly illustrates that VDBs sources are not utilised in defining its attack actions, mutual knowledge is shared with security patterns. In fact, misuse patterns complement security patterns to stop a specific attack. Although, to date, there is no well-supported evidence of their practical use as further described in Table 5. This demonstrates that misuse patterns have certain construction deficiencies and lack considerable usages from software developers' perspective, such as undefined attack pattern knowledge sources and less practical values.

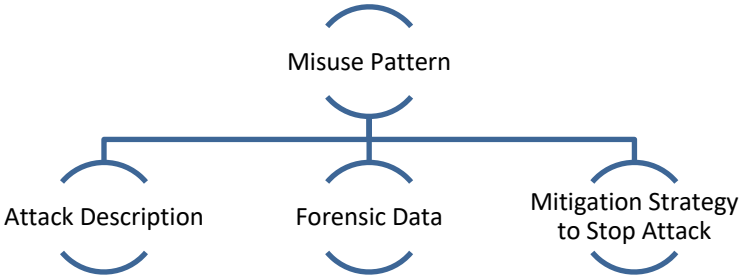


Figure 9 Anatomy of misuse pattern

Key issues	References
• **Lack of basic structure and standard format** • **No association with VDBs**	(Fernandez, Yoshioka and Washizaki 2010, Fernandez, Yoshioka, and Washizaki 2009)

Table 5 Misuse pattern key issues

3. Anti-Pattern

As discussed above in Section 2.10.2.2, in the late 1990's the Anti-Pattern concept originated, which are summarised in Figure 10. This field lacks up-to-date knowledge restoration. Despite the fact that the industry has been creating and employing anti-patterns since the invention of programmable computers (see Table 6), there is no recent literature evidence of its usage to capture vulnerabilities.

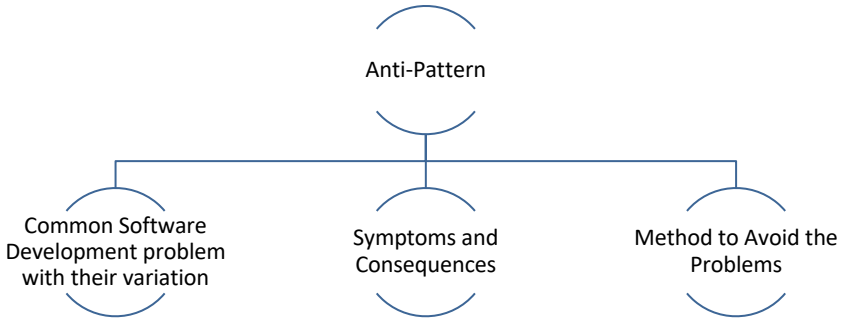


Figure 10 Anatomy of anti-pattern

Key issues	Reference
• **No association with VDBs** • **Lack of up-to-date structure and standard format**	(Foote and Yoder 1997, Dias e Silva 2014, Brown et al. 1998)1998).

Table 6 key issues of anti-pattern

4. SFP (Software Fault Pattern)

As mentioned above in Section 2.10.2.2, the SFP describes the faulty computation, which is summarised in Figure 11. Although SFP was primarily designed by security experts to automate the CWE (Vulnerability Database) intelligence in the form of patterns, a lack of detailed information about the structure and format of SFP presents a considerable obstacle for software developers. Potential shortcomings concerning developers' understanding are listed in Table 7.

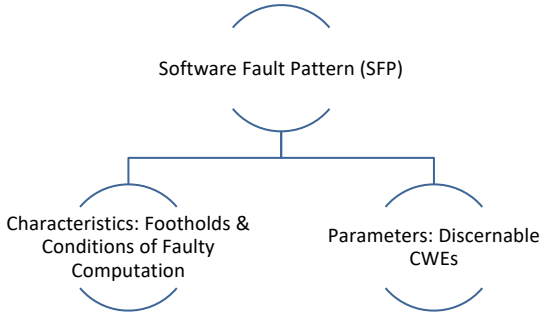


Figure 11 Anatomy of SFP

Key issues	References
• **Absence of configuration and standardization** • **Difficult to understand and implement** • **Complicated structure** • **poor communication with software engineering community**	(Mansourov 2011)

Table 7 Key issues of SFP

5. AP (Attack Pattern)

As mentioned above in Section 2.10.2.2, the AP presents attack vectors, which are summarised in Figure 12. However, AP primarily intends to present the vulnerability exploitation, it lacks a standard formulation. Literature research found that attack patterns are rarely used by software developers' due to their inherent complexity as shown in Table 8, additionally AP tend to be written in a format that is not easy for developers to understand.

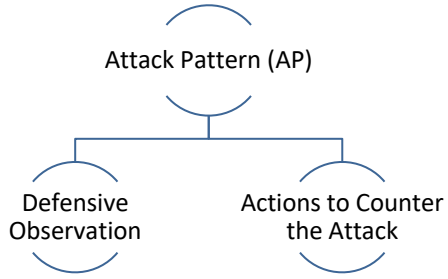


Figure 12 Anatomy of AP

Key Issues	Reference
• **Difficult to understand and implement** • **Complicated structure** • **poor communication with software engineering community**	(MITRE Corporation 2015c, MITRE Corporation 2013, Blackwell and Zhu 2014b)

Table 8 Key issues of AP

2.14 Software Engineering Community Problems: Transferring knowledge from cybersecurity to Software Engineering

As mentioned previously in Section 2.10.1, the software engineering community has been trying for a long time to integrate security, and to crack security challenges of software development processes. Each effort has separate shortcomings.

2.14.1 Continuous Efforts to Improve Libraries, Implementation Languages, and Language Processors

As mentioned above in Section 2.10.1.1, programming languages and their runtime environments are continuously being refined in order to deal with ongoing security challenges. Operating systems (OS) vendors are maintaining the pace on sustaining core operating systems security. Software engineering community has been developing in-house and add-on approaches to deal with the security posture of systems. There is a whole industry that has been trying to implement and operate security in cyberspace for some time, and it is important to understand interdependencies between software and their runtime environments.

Because it became evident that everything is relative such as computer systems, running software and operating runtime environments. It is reasonable to say that failure and success of a software system security have been highly influenced by the running environments such as OS, using programming languages such as C/C++, and their libraries.

For example, when the C runtime library (CRT) was first created over three decades ago, the security issues for computers was different as mentioned above in Section 2.2.1. However, computer systems incredible level of interconnectedness has created a vast threat environment in

cyberspace, hence escalating threats. The first question that probably comes to mind is “Why are these persistent efforts not sufficient to fulfil the security holes?” The answer is simple: standalone attempts to retrofit the programming languages and their runtime environment in order to stop security erosion of the system are not enough. Indeed, this requires a high level of collaboration and understanding between developers and security experts.

2.14.1.1 Continuing Challenges

In order to address security concerns as a part of software development processes, detailed analysis of the software engineering literature (Section 2.10.1) revealed the following particular issues. The main reasons why these efforts are not enough to provide a sufficient level of security; what preventive measures they lack against prevalent vulnerabilities are listed below:

- Deprecated & Banned functions still in use
- Legacy systems
- Developers lack the up-to-date awareness about update vulnerabilities

Key Issues	References
Deprecated & Banned functions still in use	• Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') • Security Development Lifecycle (SDL) banned function calls (MITRE Corporation 2015, Howard and Lipner 2011)
Legacy systems	• System assurance: beyond detecting vulnerabilities (Mansourov and Campara 2010)
Developers lack the up-to-date awareness	• You Get Where You're Looking For: The Impact of Information Sources on Code Security • Rethinking SSL development in an amplified world • Toward Formalization of Software Security Issues • Bridging the gap between software development and information security

	• Cyber-patterns: Unifying Design Patterns with Security and Attack Patterns (Acar et al. 2016, Fahl et al. 2013, Van and McGraw 2005, Blackwell and Zhu 2014b)

Table 9 Issues with software engineering community efforts to improve libraries

1. Deprecated & Banned Functions Still in Use

Deprecated functions or dangerous function are commonly used in developers' practices regardless of their vulnerable behaviour. It seems that developers' awareness of these functions is questionable.

Questions that need to be asked, are whether developers are aware of these functions misused or unsafe behaviour or whether they are aware of these safe alternatives to the functions.

As mentioned in Section 2.10.1.1, C/C++ programming language researchers and industrial practitioners have been proposed and developed a list of the unsafe functions that should be replaced or be re-architected to avoid C/C++ related vulnerabilities as part of secure software practices. Developers may be not aware of these alternative functions. If used properly, these do not directly pose a security risk but can introduce a weakness if not used incorrectly. These are regarded as potentially dangerous functions. A well-known example is the *strcpy()* function, in which, if a destination buffer size is provided which is larger than its source size, *strcpy()* will not overflow or cause a vulnerability. However, the misuse of *strcpy()* is so common in developers' practices that some companies prohibit *strcpy()* use entirely (MITRE Corporation 2015d). Similarly, in the case of PHP deprecated functions, which are mentioned in Section 2.10.1.1.

2. Legacy Systems

There are a large number of legacy systems, which represent enormous commercial value, and therefore their life span has been extended and enhanced to accommodate new market requirements and governmental regulations (Russinovich and Solomon 2009). When legacy systems were developed, these systems security requirements were more relaxed,

based on the not at the time deprecated and unsafe version of libraries functions.

For example, all those web-based systems, which are developed based on using earlier versions of PHP 5.3.0, due to use of deprecated functions, are now vulnerable and ready for attack. This creates a serious problem to maintain and provide security in legacy systems. As developers have previously overlooked these security systems and very often apply quick fixes based on the partial investigation are used to make them secure. Essentially, this add-on fix aggravates the problem and further comprises the security of the system (Mansourov and Campara 2010). This raises the big question of how we should address the security postures of such systems.

3. Developers Lack Up-To-Date Awareness

This research accentuates the realisation that in many cases developers' naively copy-paste insecure code obtained from internet resources (Van and McGraw 2005, Blackwell and Zhu 2014b). This highlighted a number of reasons explaining why developers do not very often use the official documentation or up-to-date resources. Arguably, official sources are less accessible in the comparison of other solutions and difficult to understand. Stack-overflow is commonly used instead of CWE (Acar et al. 2016). The issue is related to poor usability (Hans 2010) discussed in Section 2.3.1 usability. In addition, developers working under in a time constraint and under economic pressure choose resources that are easy to use and access. Software developers lack awareness because they have not been educated in this regard (Fahl et al. 2013). Developers may be unaware that those functions, have been deprecated because of their vulnerabilities, or that safe alternatives have been introduced. These practices are especially prevalent in those software developers' practices who are unfamiliar with secure coding practices (Dimitrov 2016).

2.14.2 Issues with Building Security into the Software Development Processes

During the 1990's researchers argued that security needs should be merged into the software development process and not considered an add-

on endeavour. The literature on building security into Software Development Processes and their relevance to implementing security are analysed. The literature review identified critical issues, which are summarised in Table 10. In particular, commonly used software development practices do not include security as an integral part of the software development process as it costs too much money and consumes a lot of time (Banerjee and Pandey 2009).

Critical analysis: Issues with Building Security into the Software Development Process		
Software Development Process	**Microsoft SDL**	• Risk management and compliance • Experiences threat modelling at Microsoft • System and methods for detecting software vulnerabilities and malicious code (Brenner 2007, Shostack 2008, Lomont and Jacobus 2014)
	OWASP CLASP	• On the secure software development process: CLASP, SDL and Touchpoints compared • On the secure software development process: CLASP and SDL compared (Gregoire et al. 2007, De Win et al. 2009)
Poor up-take of security		• How Do They Do It? A Look Inside the Security Development Lifecycle at Microsoft • On the secure software development process: CLASP, SDL and Touchpoints compared • Gregoire, Johan, 2007, On the secure software development process: CLASP and SDL compared • System and methods for detecting software vulnerabilities and malicious code • Risk management and compliance (Howard 2005, De Win et al. 2009, Gregoire et al. 2007, Lomont and Jacobus 2014, Brenner 2007, Shostack 2008)

Table 10 Critical analysis of building security into the software development process efforts

1. Microsoft Security Development Lifecycle (SDL)

As mentioned in Section 2.10.1.1, SDL, proposed by Microsoft, is the initiator, attempting to integrate security into the development lifecycle. However, Microsoft SDL arguably appears complicated for developers in term of understanding to implement and follow guidelines

Firstly, Microsoft SDL is not free; developers require special training to use Microsoft SDL, which means it increases the software development and maintenance cost.

Secondly, Microsoft SDL compliance with ISO 27001 (Brenner 2007), which makes it complicated in terms of threat modelling, requires a specialised security experts' team. Consequently, this is challenging for small software companies with fewer expert developers (Shostack 2008). In addition, standard compliance does not necessarily result in the elimination of vulnerabilities. The reduction of vulnerabilities is predominantly based on awareness and developers' education about how to build secure software (Lomont and Jacobus 2014). For example, the Microsoft SDL does not embed any source knowledge from cybersecurity experts, such as VDBs, which can be challenging for those software developers with limited awareness and understanding of the security vulnerabilities to apply the security guidelines effectively.

Furthermore, this has imposed paradoxical enforcement on software developers for complying with security standards regardless of developers' lack of security issues understanding and knowledge of how to prevent vulnerabilities. The organizational emphasis of Microsoft SDL may also be of limited applicability in the informal world of cross-platform application deployment.

2. OWASP CLASP

OWASP developed a process referred to as CLASP (Comprehensive, Lightweight Application Security Process) as discussed in Section 2.10.1.1. This consists of a set of independent activities that have to be integrated into the development process and its operating environment. The choice of the activities to be executed, and the order of execution, is left open for the sake of flexibility. Essentially, the execution frequency of

activities is specified per individual activity and the coordination and synchronization of activities are therefore not straightforward. (De Win et al. 2009).

OWASP CLASP implementation is limited to web-based systems. For example, like Microsoft SDL, CLASP implementation also requires a security advisor, which is not cost-effective and reasonable for small scale companies (Gregoire et al. 2007).

3. Poor Up-Take of Security

Software developers generally consider security an add-on feature (Howard 2005, Poller et al. 2017). In reality, it is not a simple feature that developers can add to a system at any phase of the development process (McGraw 1999). For example, the fact that 80% of 1998's CERT alerts involved buffer overflow problems emphasizes the point referred to Section 2.3.2.2. There is no reason that any code today should be susceptible to buffer overflow problem, yet they remain the biggest source-code security risk today (OWASP, 2015; Allen et al., 2001; Park et al., 2010; McGraw and Viega 2000). The question which needs to be addressed is: **"what we can do to educate software developers, so they can understand security errors with knowledge of how to mitigate them?"** It is suggested that security should be considered as a property as part of a complete system rather than considering it to be a bolt-on feature.

2.15 Cybersecurity Community Problems: Pushing Knowledge from Cybersecurity to Software Engineering

Attempts of the cybersecurity community to share their knowledge with software engineers (see Section 2.10.2.1) also have a number of shortcomings which create a big knowledge gap.

2.15.1 VDBs Issue

The detailed analysis of existing VDBs literature and their efficacy, particularly in order to transfer vulnerability knowledge for software developers revealed the following key issues, which are summarised in Table 11.

As discussed in Section 2.10.2.1, cybersecurity experts attempt to catalogue vulnerabilities in the form of Vulnerability databases (VDBs). These VDBs have retained a wealth of security issues and their exploitation. However, it is clear that the intended audience for these databases is not software engineers involved in developing software but rather systems administrators looking to secure their existing systems. It might be possible that the information is simply not generalized enough to be directly relevant to software developers during the development process. Literature analysis with vulnerability databases' effectiveness which was studied in order to understand software developers' vulnerability knowledge reflected upon the following issues. These are enumerated in Table 11.

Issues	Description
Lack Standardization	No standard taxonomy/classification scheme for existing VDBs, thus each of them uses their own approach, none of which were explicitly designed to use during SDLC. As such, these VDBs can typically appear complex and ambiguous to the software developer (Hui et al. 2010, Huang et al. 2013, Dimitrov 2016).
Limited knowledge	Closed source VDBs, such as the Carnegie-Mellon US Cert database and Secunia, are of necessity limited in the information that they can show concerning code-level errors (Carnegie Mellon University 2015, Secunia 2015).
Complex knowledge	It is clearly shown by many research studies, which have compared vulnerability information across the multiple VDBs that these repositories are deficient in providing interoperability, knowledge consistency and are not following standard classification schemes

	(Ghani et al. 2013, Chen, Zhang and Chen 2009, Yun-hua and Pei 2010).
Inadequate Automated tools	Researchers have tried to address VDBs complexity issue while creating a number of different tools which designed to automate the detection of vulnerabilities during development phase such as use of VDBs knowledge via text mining bug databases (Wijayasekara, Manic, and McQueen 2014, MITRE Corporation 2004, Liu and Zhang 2011, Yun-hua and Pei 2010). Nonetheless, it is clearly evident from the literature that problematic ambiguities still exist in their classification strategies.

Table 11 Issues of VDBs usability for developers

2.16 Conclusion

90% of security incidents result from exploitation of flaws in software systems (DHS 2017). In reality; however, software developers struggle against recurring and consistent software flaws (i.e. buffer overflows, and integer overflows), which are exploited on a daily basis by malicious hackers. Questions arising from this are:

- Why do malicious hackers know more about our systems than developers and security experts?
- What are the knowledge gaps or disconnects between developers and security experts that allow malicious hackers to succeed?

Nonetheless, a large body of knowledge about software vulnerabilities exists within the cybersecurity community, in particular amongst penetration testers and ethical hackers. Currently, cybersecurity experts put much effort into classifying discovered vulnerabilities and developing taxonomies of these vulnerabilities. Such vulnerabilities are then catalogued in publicly available vulnerability databases (VDBs).

Similarly, software developers have worked to embed security within the software development process in order to improve the security of software systems. Various attempts to capture and formalise the transferring

knowledge in a manner appropriate to software engineers have been made such as improving libraries C/C++ and PHP, introducing secure development lifecycle (Microsoft SDL, and OWASP CLASP), and including Misuse Patterns, Software Fault Patterns (SFP), and Security Patterns (SP).

Despite the software engineering community's best efforts, the number of dangerous software exploitations is increasing at an alarming rate. It is thus clear that these resources are not being utilised effectively in providing the necessary knowledge to the software developer due to the following reasons: information overload, lack of techniques to systematically annotate flaws' rectification and insufficient analysis of the data relating to these prevalent vulnerabilities. In fact, other impediments include the knowledge transfer mechanisms between the work on vulnerability databases (VDBs), developers' perceptions of security issues and the complexity of the Microsoft security development lifecycle (SDL) is complex, which creates a distinct communication gap between cybersecurity experts and software engineers. Interruption of (knowledge) communication directs software developers to repeat practices that lead to vulnerabilities and gives rise to software flaws exploitation. The need for a better understanding of vulnerabilities via a usable knowledge transfer mechanism and our proposed solution is the subject of the remainder of this dissertation.

Contribution

This section of the thesis comprises of three chapters, chapter 3 derivation of Vulnerability Anti-Patterns (VAPs), chapter 4 Vulnerability Anti-Patterns (VAPs), and chapters 5 creating the catalogue of VAPs.

Chapter 3 critiques existing pattern-based approaches and define the derivation process of VAPs. This chapter describes the motivation and context for investigating and addressing the following problem. VAP can bridge the security knowledge gap faced by developers while successfully transferring the usable security information from the cybersecurity community to software developers. The goal of VAP development is to provide security awareness of flaws that lead to vulnerabilities.

Chapter 4 elicits the main work product VAP and its design structure. This chapter explains the VAP types which will use during the evaluation process.

Chapter 5 describes the created catalogue of VAPs, which comprises of 9 “Informal” and 11 “Formal” VAPs.

3 Criticism of Existing Pattern-Based Approaches and the Derivation of a New Approach: Vulnerability Anti-Pattern (VAP)

This chapter introduces the methodology used for evaluating the contribution of this dissertation. The research examines how the cybersecurity and software engineering communities' deal with the concept of poor development practices and vulnerabilities based on the use of patterns as a research topic. Informed by these different approaches, this chapter describes an approach based on Anti-Patterns to use within this dissertation.

The key objectives of the chapter are:

- Section 3.1 provides an overview and criticism of current pattern-based security research.
- Section 3.2 performs the comparison Analysis of selected Cyber-Patterns against Vulnerabilities
- Section 3.3 describes a proposed methodology.
- Section 3.4 describes steps engaged in capturing vulnerability knowledge in the form of vulnerability anti-patterns.

3.1 Pattern-Based Research Approaches

An analysis of both software engineering and cybersecurity communities' pattern-based approaches has performed. This includes the knowledge sources that are required to design a methodology for an effective transfer of vulnerability knowledge to improve software developers' awareness. Pattern-based approaches originating in the cybersecurity domain are considered in Section 3.1.1, followed by those from the software engineering community in Sections 3.1.2 and 3.4.1, leads to the derivation of VAPs.

To deal with the inadequacies of existing pattern-based approaches, while fulfilling developers' needs to understand vulnerabilities and their root-causes, specific vulnerability knowledge is obtained from each selected pattern-based approach to provide an optimal solution.

3.1.1 Cybersecurity Pattern-Based Research Methods

In cybersecurity, pattern-based research tends to focus on evaluating the exploitation of vulnerabilities or misuse of errors, rather than investigating why the errors occur or why errors constitute vulnerabilities. This position is based on the assumption that poor software development practices will stop if usable and understandable knowledge is provided, and the appropriate knowledge transfer methods are used to help software developers during Software Development Lifecycle (SDLC) as shown in Figure 13. Two pattern-based approaches related to security have been carefully chosen from the cybersecurity community for this purpose: Software Fault Patterns (SFP) (Mansourov 2011) and Attack Patterns (AP) (MITRE Corporation 2014)

3.1.1.1 Software Fault Pattern

Common Weakness Enumeration (CWE) and s catalogue of weaknesses provide for the most commonly known open source vulnerability database. SFPs are associated with the CWE database, which provides a formal specification of weaknesses (vulnerabilities). As mentioned in Section 2.13, the CWE restructuring into a cluster is known as a Software Fault Pattern that aims to make a vulnerability database such as CWE more understandable and easier to use. However, a lack of detailed information about the structure and format of SFP posits a considerable obstacle for software developers in understanding them (Mansourov 2011).

3.1.1.2 Knowledge Sourced from Software Fault Pattern

From SFP the useful set of information is considered based on the need of software developers in order to provide with an understanding of vulnerability, its root-cause and traceability in the code. The derivation process of VAPs includes:

a) **White-box Vulnerability Pattern**

This provides transparency to trace a vulnerability and its properties (c.f. white-box testing).

b) **Footholds**

This draws attention to vulnerability characteristics during the development of system artefacts such as code, database schemas and platform configuration.

c) **Code path (the condition of this pattern)**

Necessary conditions for the connecting elements of the code to find coding errors/mistakes.

3.1.1.3 Attack Pattern

An Attack Pattern is considered analogous to a design pattern, which describes how a particular type of attack is performed as an abstract pattern (Blackwell and Zhu 2014a p.115). As explained in the literature review chapter Section 2.10.2.2. CAPEC (MITRE Corporation 2015c) introduced the concept of Attack Patterns to describe the attack procedure. However, it lacks the standardisation of design patterns. Consequently, software developers do not use Attack Patterns as they usually appear to be complicated and difficult to understand (Faily, Parkin and Lyle 2014). It is also apparent from the literature that adoption of attack patterns is not common in software developers' development practices due to their inherent complexity and lack of standardisation, but also because attack patterns, do not tend to be written in a usable format according to software developers' needs (Section 2.13).

3.1.1.4 Knowledge Sourced from Attack Pattern

From Attack Pattern the useful set of information is considered based on the need of software developers in order to provide an understanding of vulnerability, it includes an overview of attack and its formats:

a) Pattern Name and Classification

A unique, descriptive identifier for the pattern.

b) Attack Prerequisites

What conditions must exist or what functionality and what characteristics must the target software have, or what behaviour must it exhibit, for this attack to succeed?

c) Description

A description of the attack including the chain of actions taken.

d) Related Vulnerabilities or Weaknesses

What specific vulnerabilities or weaknesses does this attack leverage?

e) Method of Attack

What is the vector of attack used (e.g., malicious data entry, maliciously crafted file, protocol corruption)?

f) Attack Motivation-Consequences

What is the attacker trying to achieve by using this attack?

g) Attacker Skill or Knowledge Required

What level of skill or specific knowledge must the attacker have to execute such an attack?

h) Resources Required

What resources (e.g., CPU cycles, IP addresses, tools, time) are required to execute the attack?

i) Solutions and Mitigations

What actions or approaches are recommended to mitigate this attack, either through resistance or through resiliency?

j) Context Description

In what technical contexts (e.g., platform, OS, language, architectural paradigm) is this pattern relevant?

k) References

What are further sources of information available to describe this attack?

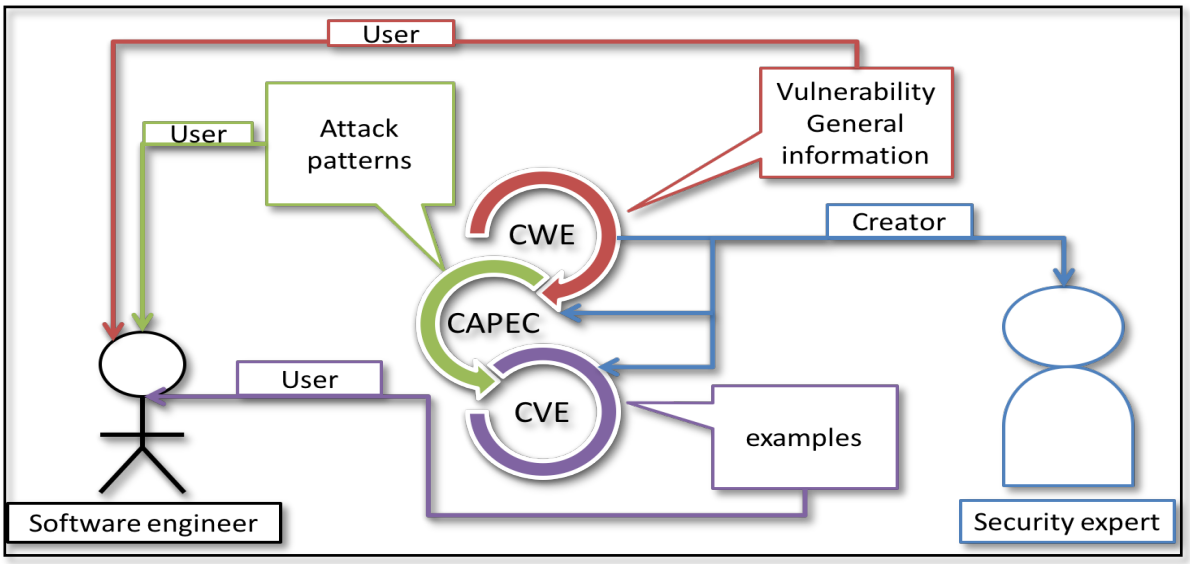


Figure 13 Relationship between pattern-based approaches and cybersecurity & software engineering experts

3.1.2 Software Engineering Pattern-Based Research Methods

This section presents software engineering pattern-based research methods as discussed in Section 2.10.2 included in this thesis are: Security Patterns (SP) (Schumacher et al. 2013) and Anti-Patterns (Brown et al. 1998).

3.1.2.1 Security Pattern

SP defines a solution against a set of specific threats which are controlled via applying a specifically designed security mechanism. Moreover, SP defines a solution, rather than directly revealing the underlying cause of software errors (Schumacher et al. 2013).

3.1.2.2 Knowledge Sourced from Security Pattern

From a security pattern, the useful set of information is considered based on the need of software developers in order to provide an understanding of vulnerability. The useful set of information of SP that is considered for developers in this research includes:

a) Example

Problem situation where the use of this pattern may provide a solution.

b) Context

Relevant context and its characteristics in which a solution is applicable.

c) Problem

It indicates the risk forces that affect the possible solution.

d) Solution

Describes the idea of the pattern.

e) Structure

The static view of the solution and some dynamics aspects in the form of a sequence diagram.

f) Dynamic

UML diagrams such as class diagram, sequence diagram and use cases.

g) Implementation

Set of recommendations when or where to do to use this pattern.

h) Example resolved

Following contents of the above pattern will lead to resolutions.

i) Consequences

The benefits and liabilities of the solution embodied in this pattern.

j) Known uses

Minimum three examples of its use in a real system.

k) See also

Related other known patterns.

3.1.2.3 Anti-Pattern

An Anti-pattern is a mechanism to describe poor development practices to a developer that have the potential to generate significant negative consequences. Anti-Patterns examine the causes, symptoms, and consequences of poor software development practices and in return, offer a refactored solution that provides a successful solution (Brown et al. 1998).

3.1.2.4 Knowledge Sourced from Anti-Pattern

The useful set of information of anti-pattern that is considered for developers in this research includes:

a) Name

The key problem name is addressed in this Anti-pattern.

b) Also known as

This identifies an additional popular, descriptive name and phrases for the Ani-pattern.

c) Most frequent scale

This section identifies where this anti-pattern fits into predefine scale SDLM

- Global
- Enterprise
- System
- Application
- Framework
- Micro-architecture
- Object

d) Refactored solution name

The refactored solution patterns name is a key reference, particularly when paired with the Anti-pattern name for the problem.

e) Refactored solution type

This will identify the type of improvement that results from applying the Anti-pattern solution:

- Software, involving the creation of new software
- Technology, solving the problem by the adoption of technology

- Processes, providing the definition of activities that are consistently repeatable
- Role, allocation of clear responsibilities to organizational stakeholders.

f) Root causes

The general causes for this Anti-pattern.

g) Unbalance forces

The identifiers the primal forces that are ignored, misused or overused in this Anti-pattern.

h) Anecdotal evidence

Common phrases and humorous anecdotes that succinctly describes the problem.

i) Background

This section briefly describes the context of the problem.

j) General form

General characteristics of the Anti-pattern are identified and an overview of the nature of the problem is presented.

k) Symptoms and consequences

It provides a list of symptoms that direct to the resulting consequences.

l) Typical causes

The specified problem causes of this Anti-pattern.

m) Known expectations

Potential cases in which the usage expectations can occur.

n) Refactored solution

It resolves the symptoms and consequence, typical causes and unbalances forces issues.

o) Variation

It has shown the variation in known expectations.

p) Example

A real-world experience which provides the refactored solution.

q) Related solution

Other related solution against this Anti-pattern.

r) Applicability to other viewpoints and scales

A viewpoint of users or an appropriate scale is described.

3.2 Comparison Analysis of Cyber-Patterns against Vulnerabilities

A comparative analysis of the selected types of patterns related to a security described in Section 3.1.1 to Section 3.1.2 is investigated in Table 12. The literature review of existing cyber-patterns related to cybersecurity or software engineering communities and highlighted the deficiencies of transfer vulnerability knowledge. In addition, Table 12 explicates the existing cyber-patterns inadequateness including the vulnerability databases issues as mentioned previously 2.15.1. Consequently, it is beneficial to do a comparison analysis of existing pattern-based approaches; subsequently, we can find and design an effective solution.

The headings used in Table 12 are discussed below:

- **Pattern Name:** pattern original name
- **Context:** Pattern implementation perspective
    - **Software developers**
    - **Cybersecurity experts**
- **Usability:** Usability in the context of providing understanding to software developers range from Poor-high
- **Security Concerns:** pattern enactments to the security of the system and answer will be in yes or no
- **Functionality concerns:** pattern ratification of the system requirements and the answer will be in yes or no
- **Target Audience:** The main user of the pattern such as software developer and security experts
- **Knowledge Source Community:** cybersecurity community or software engineering community
- **Developer Expertise Level:** Developers require expertise to use this pattern and is divided into three levels: low, medium, high

		Pattern Name	Context	VDBs Knowledge Association	Usability	Security Concerns	Functionality Concerns	Target Audience	Knowledge Source Community	Developer Expertise Level
Cybersecurity Pattern-based Approaches		**Software Fault Pattern**	Faulty Computation Template	Yes	Poor	Yes	No	Security Experts	Cybersecurity Community	High
		Attack Pattern	Attack Mechanism Template	Yes	Poor	Yes	No	Security Experts	Cybersecurity Community	High
Software Pattern-Based Approaches	engineering	**Security Pattern**	Security Mechanism Template	No	Poor	Yes	No	Software Developers	Software Engineering Community	High
		Anti-Pattern	Refactored Solution Template	No	No	No	No	Software Developers	Software Engineering Community	Low-high

Table 12 Analysis Table of existing pattern-based approaches and their relation to capture and transfer vulnerability knowledge

3.3 Improved Use of Pattern-Based Approaches to Capture Poor Software Development Practices (Vulnerabilities)

Software developers are generally trained to develop software systems in an ideal scenario, in contrast to security experts (ethical hackers or pen testers) who generally investigate exploitation mechanisms of software systems (Van and McGraw 2005).

As mentioned previously Section 2.15.1, the complicated structure of vulnerability databases (VDBs), and their inadequacies in capturing and transferring vulnerability knowledge via existing pattern-based approaches pose major challenges to software developers in order to understand the root causes of vulnerabilities (Hans 2010, Nafees et al. 2017).

Furthermore, cyber-patterns are ineffective in intimating a common understanding with adequate communication among software engineering and cybersecurity communities' experts (Blackwell and Zhu 2014b). As a result, Software developers repeatedly commit common development errors. As discussed in the literature review chapter (Sections 2.10.2 and 2.11) the research concludes the following potential reasons:

- Lack of intuitive knowledge communicated from vulnerability databases (VDB).
- Common use of outdated knowledge sources and transfer mechanisms.
- Insufficient awareness of development errors and their underline root causes that lead to vulnerabilities.

To address this problem, this work proposes a means of expressing vulnerability knowledge in the form of anti-patterns to provide guidance and assistance to software developers.

The concerns this thesis raises are: how can patterns encapsulate essential understanding of security related vulnerabilities and how can patterns effectively present the exploitation of errors to developers (Yun-hua and Pei 2010).

The contention between raising vulnerability awareness of developers and transfer of security knowledge is paramount to success as an effective way of capturing and transferring of vulnerability knowledge, in comparison to existing efforts such as security patterns, attack patterns and software fault patterns that failed to do so. This study performs a comparative analysis among selected vulnerability related cyber-patterns to evaluate and identify the reasons why existing cyber-patterns are

ineffective in providing useful knowledge of vulnerabilities. The literature review Section 2.10.2.2 explained patterns in detail. Table 13 provides a brief overview of existing cyber-patterns' ineffectiveness, which is based on the criteria of their availability, understanding and usability for developers. CIA impact and mitigation techniques (MITRE Corporation 2015b) are commonly used within the cybersecurity industry, to transfer usable knowledge of the vulnerability, we have chosen three main categories: vulnerability knowledge availability while providing in a format that is understandable and useful for developers.

Table 13 main headings description are:

- **Availability:** Lack of accessibility and its usage information. For example, software fault patterns are less accessible in comparison to the design pattern.
- **Understanding:** Lack of a standard format makes it difficult to understand. For example, attack patterns lack the standard format of representation.
- **Usability:** Lack of usable knowledge format sourced from Cybersecurity community. For example, security patterns lack communication with vulnerability knowledge sources.

The following table provides pattern content information. Such as

- **Mostly:** Publicly available, easy to understand, easy to use
- **No:** Not publicly available, difficult to understand, difficult to use.

Pattern Type	Availability	Understanding	Usability
Design Pattern	Mostly	Mostly	No
Security Pattern	Mostly	No	No
Software Fault Pattern	No	No	Mostly
Attack Pattern	Mostly	No	No
Anti-Pattern	No	Mostly	No

Table 13 Comparative analysis of cyber-patterns to measure ineffectiveness in order to vulnerability understanding

3.4 Proposed Solution

From this review of existing pattern-based approaches to security, it can be concluded that any new approach needs to deal with the following considerations:

- The new method needs to use cybersecurity knowledge sources. However, knowledge extracted from cybersecurity sources should be presented in a usable format which software developers can understand.
- The new approach needs to contribute a new pattern-based approach that can be used to identify and capture any vulnerability within the Software Development Lifecycle (SDLC) and provide mitigation solutions.

The pattern-based approach proposed in this dissertation, which is known as "Vulnerability Anti-Pattern", is described in more detail in the following section.

3.4.1 Derivation of Vulnerability Anti-Pattern (VAP)

There are three main derivation processes of the VAP approach:

1. **Knowledge Extraction Process (KEP)**
   The KEP involves "pulling" information about vulnerability from appropriate sources, e.g. existing vulnerability databases (e.g. CWE, CAPEC, and CVE) and security patterns (SP).
2. **Knowledge Provision Process (KPP)**
   The KPP involves taking the output of KEP and storing /expressing it in the form of a VAP.
3. **Knowledge Awareness Process (KAP)**
   This is the activity of presenting the VAP to software developers for knowledge awareness of vulnerabilities.

As shown in Figure 14, the proposed methodology is divided into sub-parts to achieve simplicity and reduce complexity. The KEP in Section 3.4.1.1 explains the "Knowledge pulling", to extract knowledge from different VDBs, security patterns and attack patterns sources; The KPP in Section 3.4.1.2 describes the "Knowledge pushing", to encapsulate the normalised knowledge into anti-pattern to apprehend the necessary knowledge of vulnerabilities.

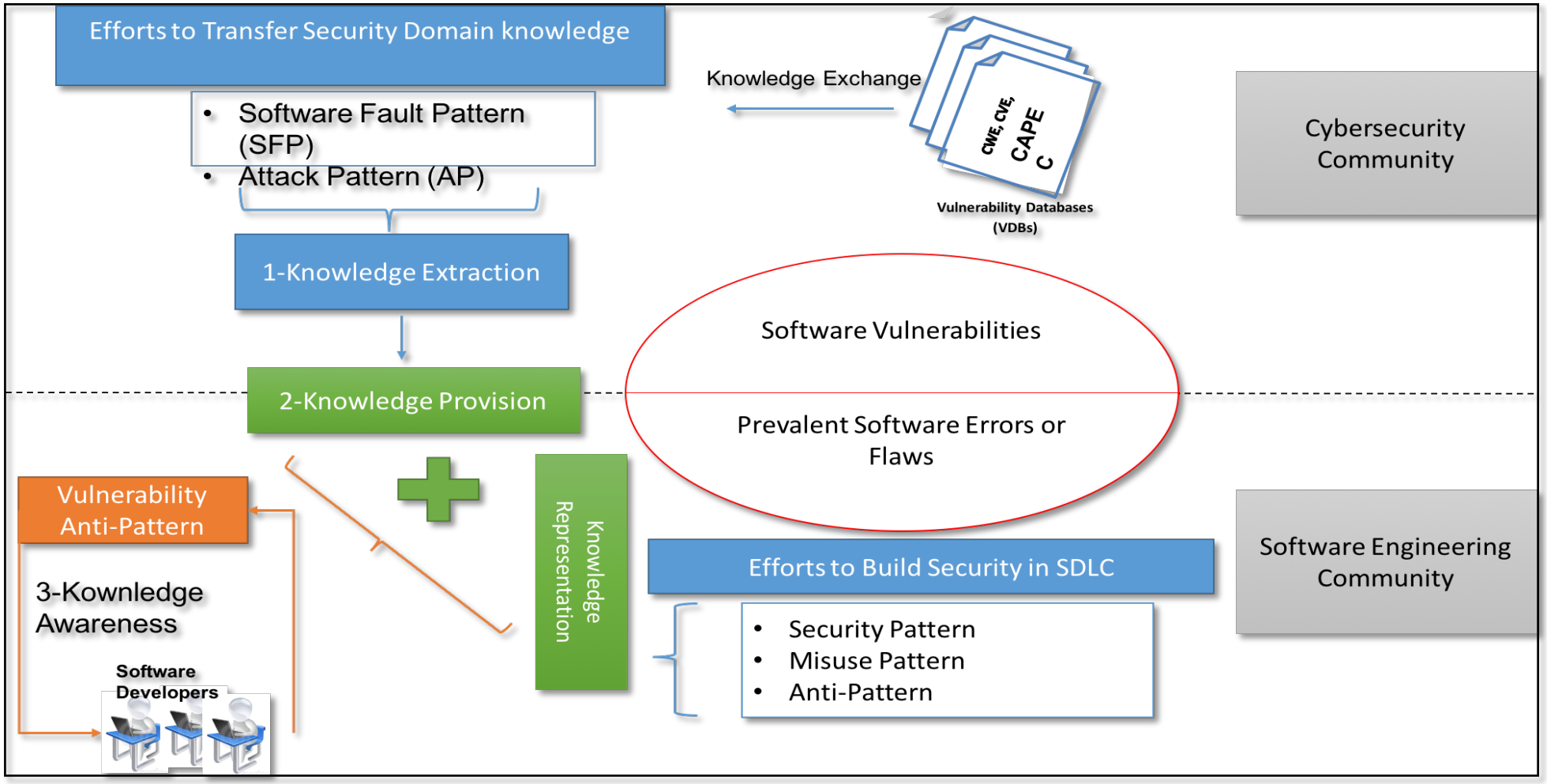


Figure 14 Derivation of VAP[3]

[3] 1-knowledge extraction is KEP, 2-knowledge provision is KPP, 3-Knowledge Awareness is KAP

3.4.1.1 Knowledge Extraction Process (KEP)

KEP is a process during which cybersecurity knowledge sources are used to extract developer centric knowledge, which is essential to improve developer awareness of how malicious hacker exploit their development errors. To follow up, KEP maps extracted knowledge into SDLC, which provides knowledge to developers about which stage of the development lifecycle the vulnerability originates and what root causes leading to the vulnerability. KEP comprises:

- Knowledge extraction
    - Create nomenclature of Vulnerabilities
- Knowledge mapping into SDLC
    - Generate a Decision tree

Knowledge extraction: During this process, knowledge is extracted from multiple sources such as VDBs (CWE, CVE), security patterns and attack pattern databases (CAPEC) as shown in Table 14, which explains the knowledge sources such as Software Fault Patterns (SFP), Design Patterns (DP), Anti-Patterns and Attacks Patterns (AP) and describes accrued knowledge from these sources. Based on extracted knowledge, a nomenclature of vulnerabilities is created.

Knowledge mapping into SDLC: After extraction, the vulnerability knowledge is mapped into the SDLC. The mapping to SDLC provides awareness to developers about which stage of development lifecycle the vulnerability originates in and what root causes leading to the vulnerability. A mapping process is used to generate a decision tree, which maps vulnerabilities to root causes within the SDLC and generates “injury” and “safeguard” paths based on extracted knowledge from VDBs and security related cyber-patterns within SDLC.

Table 14 illustrates the general overview of the knowledge extraction process and its mapping to cyber-patterns within the SDLC. This SDLC mapping of vulnerability adds value to VAP design. Basically, this approach helps developers to trace the vulnerability root cause from the initiated level within the SDLC. The research includes three main phases of the SDLC during which poor security coding practices generate weaknesses that have the potential to turn into vulnerabilities. The mapping considers the following the SDLC phases:

1) Requirement specification
2) Design

3) Implementation

Vulnerability level in SDLC	Software Fault Pattern	Vulnerability Databases	Analysis Process of Security Related Cyber-patterns
Requirement Level Vulnerabilities	Injury, safeguard	CWE	Security objectives
Design Level Vulnerabilities	Injury, safeguard	CWE, CVE	Design patterns/anti-pattern
Code/Implementation Level Vulnerabilities	Injury, safeguard	CAPEC	Secure coding/attack patterns

Table 14 Mapping between vulnerabilities and cyber-patterns within SDLC

1) **Requirement Specification Phase:** During this phase, inaccurate security requirements and any security weaknesses violating security requirements are potential security flaws. Security objectives present the solution for these security flaws. The knowledge is extracted from Common Weakness Enumeration (CWE) (MITRE Corporation 2015b), which can provide a list of security flaws that can help find and resolve vague requirements and achieving the security objectives (Alvi and Zulkernine 2011).
2) **Design Phase:** During the design phase, the wrong algorithm approach, incorrect data conversion and unsafe exception handling are design flaws which may result in vulnerabilities being introduced into the software. The knowledge is extracted from CWE and CVE, while using the Anti-Pattern approach. This creates a link between the security flaws and properties to inform upon the reasons of anti-patterns within the VAP.
3) **Implementation phase:** During the implementation phase, there are many security guidelines to be used for preventing possible security risks (Hans 2010, Shiralkar and Grove 2009). The main focus should be on a secure coding pattern to attack patterns. Common Attack Pattern Enumeration and Classification (CAPEC) (MITRE Corporation 2015c) are very helpful for linking software flaws to attack patterns.

Primarily, the KEP is designed to transfer vulnerability knowledge, which comprised of vulnerabilities nomenclature to assign developers understandable vocabulary and decision tree to find the injury (flaw leads to vulnerability) and (safe solution against flaw) safe path. This process includes:

1) Nomenclature of Vulnerabilities
2) Decision Tree

These are detailed below.

1. Nomenclature of Vulnerabilities

During the KEP, a nomenclature is generated based on the extracted knowledge that is sourced from CWE, CVE, CAPEC and security patterns. This vocabulary helps developers to understand cybersecurity experts' terminologies. The including information are:

- **Vulnerability Information:** General vulnerability information comprises ID, registered CVE vulnerable examples CVE and other general names given to vulnerability, which are sourced from CWE, and CVE.
- **Vulnerability Footprints or Characteristics:** This category includes the context in which vulnerabilities generally occur, Software Development Lifecycle phase in which vulnerability originates, and vulnerability software fault pattern to expose the faulty computation. All information is obtained from CWE and SFP.
- **Mitigation:** To find solutions against these vulnerabilities, we include possible threat information, related solution patterns, and attack patterns that are sourced from STRIDE threat model, a catalogue of security pattern and attack pattern from CAPEC.

The vulnerability nomenclature is demonstrated in Table 15 with the knowledge source information. This is used as a sanitised knowledge bank for designing the Vulnerability Anti-Pattern. During KEP, vulnerabilities nomenclature is manually generated by the researcher and updated quarterly since 2015, but as a part of future work which is described in Section 10.5.1. Automated tool will create to capture and sanitise recent vulnerability information, which will automatically update vulnerabilities nomenclature information regularly.

Vulnerability information[4]			*Vulnerability fingerprint or characteristics*			*Mitigation*		
CWE-ID	**CVE**	**Generally known as**	**Context**	**Lifecycle**	**SFP**	**STRIDE**	**SP**	**AP**
CWE-89	CVE-2016-1393, CVE-2015-0161	SQL Injection	Software fails to correctly escape special elements used in SQL commands.	Design Phase	CWE-990: SFP Secondary Cluster: Tainted Input to Command	Spoofing	Intercepting Validator	CAPEC-7, CAPEC-66, CAPEC-108, CAPEC-109 CAPEC-110
CWE-862:	CVE-2009-3168, CVE-2009-3597, CVE-2009-2282	Missing Authorization	The software does not perform an authorization check when an actor attempts to access a resource or perform an action.	Design Phase		Information Disclosure	Role-based access control	CAPEC-1, CAPEC-17, CAPEC-58

[4] Vulnerability databases searched between 15/07/2015 -12/11/2017

CWE-306	CVE-2004-0213, CVE-2008-6827, CVE-2002-1810	Missing Authentication for Critical Function	The software does not perform any authentication for functionality that requires a provable user identity or consumes a significant amount of resources.	Design phase	CWE-952: SFP Secondary Cluster: Missing Authentication	Spoofing	Authentication	CAPEC-225, CAPEC-12, CAPEC-36, CAPEC-40, CAPEC-62
CWE-120	CVE-2016-5108, CVE-2016-5108	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	The program copies data to a buffer without checking the size of the input.	Implementation phase	CWE-970: SFP Secondary Cluster: Faulty Buffer Access	Tampering	Safe Data Structure	CAPEC-8, CAPEC- 9, CAPEC-10, CAPEC-14, CAPEC-24
CWE-676	CVE-2011-0712, CVE-2009-3849, CVE-2006-2114	Use of Potentially Dangerous Function	The program uses a potentially dangerous function that may introduce a vulnerability if used incorrectly.	Design and Implementation Phase	CWE-1001: SFP Secondary Cluster: Use of an Improper API	NONE	NONE	CAPEC-113

CWE-131	CVE-2004-1363, CVE-2008-0599	Incorrect Calculation of Buffer Size	The software does not correctly calculate the size to be used when allocating a buffer,	Implementation Phase	CWE-974: SFP Secondary Cluster: Incorrect Buffer Length Computation	Tampering	NONE	CAPEC-47, CAPEC-100
CWE-190	CVE-2010-2753, CVE-2005-0102, CVE-2005-1141	Integer Overflow or Wraparound	The software performs a calculation that can produce an integer overflow or wraparound.	Implementation Phase	CWE-998: SFP Secondary Cluster: Glitch in Computation	Tampering	NONE	CAPECE-92
CWE-79	CVE-2008-5080, CVE-2007-5727	Improper Neutralization of Input During Web Page Generation	The software does not properly escape attacker-provided data when generating HTML content.	Design and Implementation Phase	CWE-990: SFP Secondary Cluster: Tainted Input to Command	Information disclosure, Tampering	Container Managed Security, Message Interceptor Gateway	CAPEC-18, CAPEC-19, CAPEC-32, CAPEC-63, CAPEC-85
CWE-352	CVE-2009-3759, CVE-2009-3520,	Cross-Site Request Forgery (CSRF)	The web application does not sufficiently verify that the source of the request is the same as the target of the request. This	Design and Implementation Phase	NONE	Spoofing	Synchronizer token pattern	CAPEC-62, CAPEC-111

	CVE-2005-1674		enables a command (triggered from a malicious application) to be sent to a trusted website using the user's browser.					
CWE-134	CVE-2007-2027, CVE-2006-2480, CVE-2002-1788	Uncontrolled Format String	The software uses formatted output functions with a format string controlled by an attacker.	Implementation Phase	CWE-990: SFP Secondary Cluster: Tainted Input to Command	NONE	NONE	CAPEC-67

CWE-78	CVE-2012-1988, CVE-2007-3572, CVE-2008-4796	Improper Neutralization of Special Elements used in an OS Command	The software does not properly escape the special elements used in an operating system command, which may enable execution of arbitrary commands by an attacker.	Design and Implementation Phase	CWE-990: SFP Secondary Cluster: Tainted Input to Command	NONE	NONE	CAPEC-6, CAPEC-15, CAPEC-43, CAPEC-88

Table 15 Taxonomy of vulnerabilities

2. Decision Tree

In the KEP, after categorisation using the developed nomenclature, the vulnerabilities are based on their extracted knowledge, Decision tree generation has two main aims: firstly, to carry out the vulnerability mapping for finding the root-causes within the SDLC, secondly, to apply the obtained VDBs and vulnerability related pattern information to indicate the “Injury” and “Safeguard” paths.

- “Injury Path” means that flaws lead to vulnerability.
- “Safeguard Path” means that the flaw can be avoided to prevent the vulnerability from occurring.

Consequently, the decision tree can depict safeguarding and injury flows which are associated with security incidents, including their low-level and high-level root causes within the SDLC phases, as shown in Figure 15.

- Green paths are safeguards that direct the developer to avoid software vulnerabilities.
- Following Red (injury) paths can lead to the creation of a vulnerability.

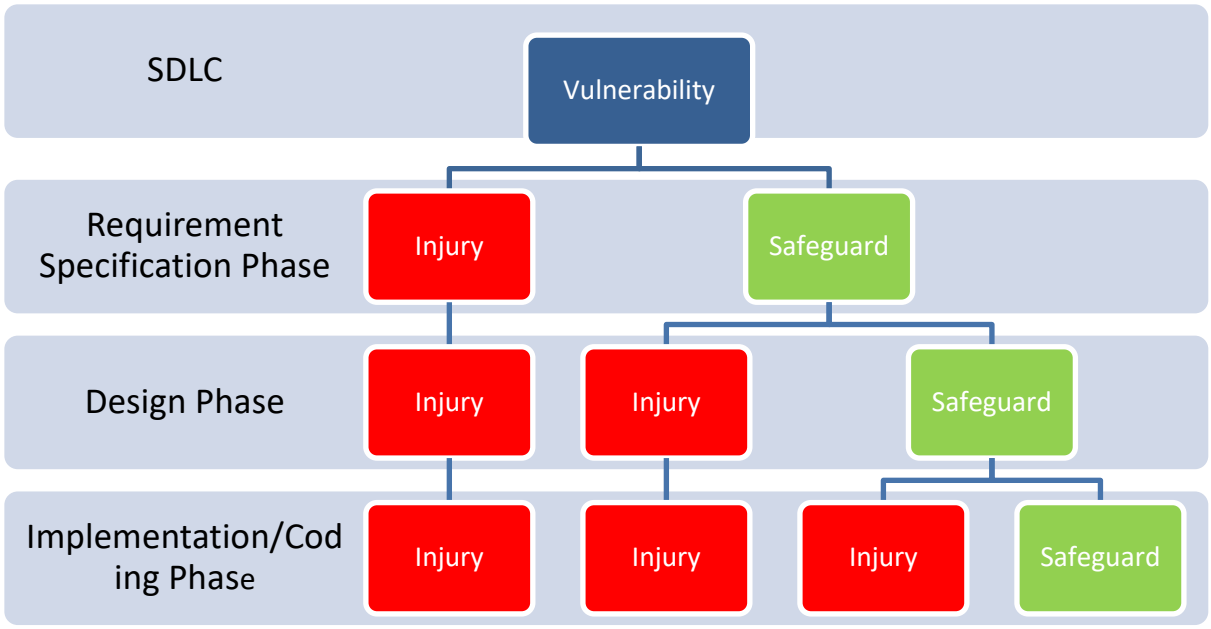


Figure 15 Vulnerability flow decision Tree in SDLC

To illustrate a decision tree, CWE-190: Integer overflow or wraparound is exemplified in the following section. This vulnerability occurs due to an incorrect logic assumption, in which larger integers are stored in a small size integer value.

i) Decision Tree Example: CWE-190, Integer Overflow or Wraparound

Figure 16 is an example of a decision tree for the *Integer Overflow vulnerability*, which is chosen from the list of OWASP "Top 25 Most Dangerous Software Errors". The CWE-190 vulnerability explanation mapping within SDLC is described below:

- **Requirement Specification Phase:** During this phase, Information is sourced from the CWE and OWASP, and VDBs, which helps us find the security objectives[5] for integer overflow vulnerability. It can conclude that careful language selection[6] is the green (safeguarding) path that performs bound checking otherwise it can lead to the injury path of numeric error (sourced from software fault pattern-CWE-998) at this stage.
- **Design Phase:** During the design phase, information passed from the above phase leads to the further subdivision. The injury path (numeric error) knowledge sourced from software fault pattern that can turn into integer overflow vulnerability and safeguard path knowledge scoured from security pattern that follows secure coding practices such as *Safeint libraries*, which is linked to a security pattern called Safe Data Structure[7].
- **Code/Implementation Phase:** The CAPEC information source helps the developer find the related attack patterns and interlinked vulnerabilities, such as related attack patterns are CAPEC- 92 forced integer overflow, and interlinked vulnerability CWE-680 integer overflow to buffer overflow that can occur due to the injury path. However, to follow the safe path, such as the bounds-checking library (gcc 2.7.0) and related security pattern (Safe Data Structure), can help to avoid introducing this vulnerability.

---

5 System requirement specifications to deal with security concerns

6 The CWE recommends (MITRE Corporation 2015c) using a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. If possible, choose a language or compiler that performs automatic bounds checking.

7 Security pattern

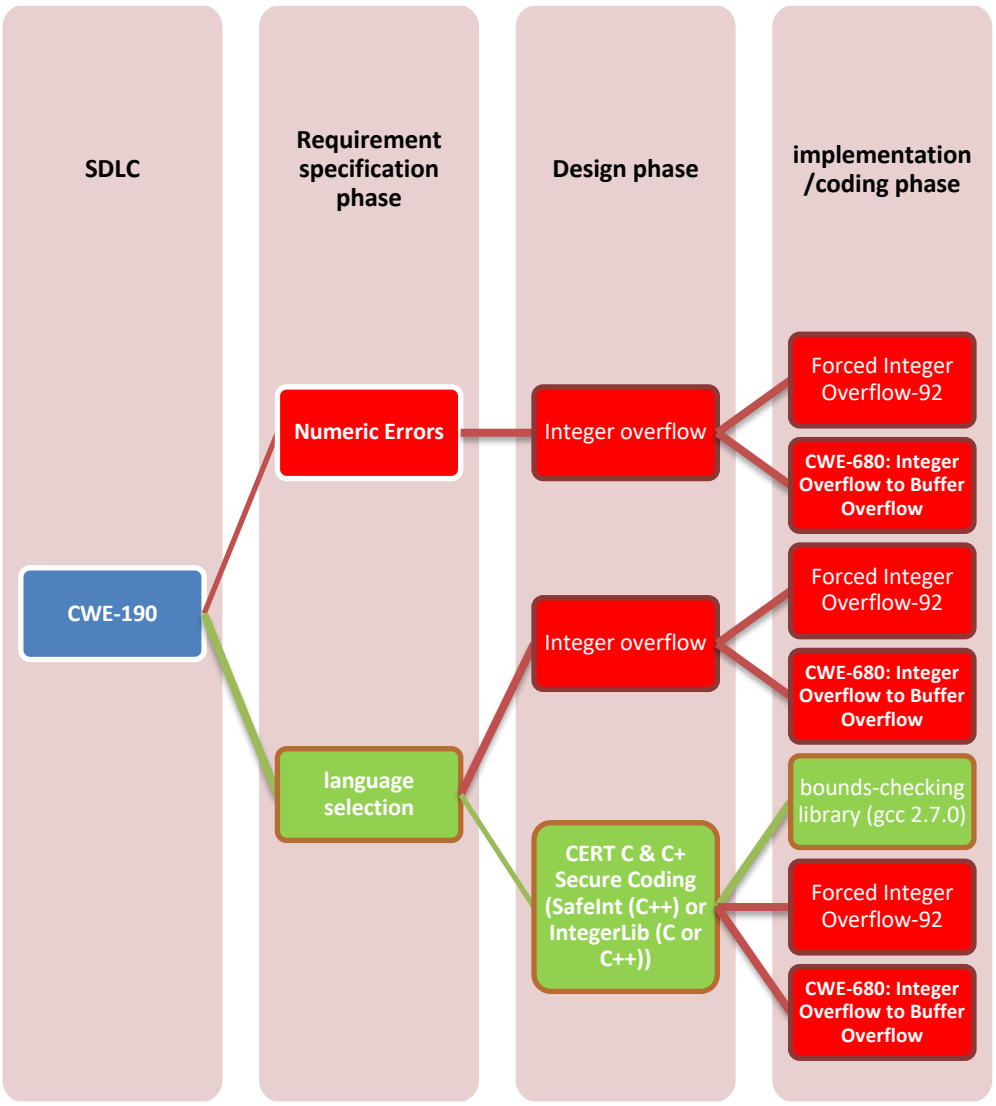


Figure 16 Decision tree mapping example: CWE-190

The point to be noted here is that an “injury” path in one phase always leads to an “injury” path in subsequent phases.

The CWE-190 vulnerability originated during the design phase due to poor requirement specification, which is confirmed with decision tree mapping. Use of safe language will lead developers to the safeguard path. Once an injury is induced, it is difficult to mitigate. For example, CWE-190 is exploited as CAPEC-92(forced integer overflow) and leads to its chain vulnerability (CWE-680) that is called interlinked vulnerability (vulnerability always triggers another vulnerability, also called a chained vulnerability). For example, integer overflow mostly causes a buffer overflow.

3.4.1.2 Knowledge Provision Process (KPP)

The Knowledge Provision Process makes use of anti-patterns to capture and integrate the extracted information which forms the output of the KEP, so that software developers have access to the distilled wisdom of cybersecurity experts in dealing with recurring development errors related to security. The extracted knowledge is pushed into the Anti-pattern to encapsulate the necessary knowledge of the vulnerability.

a) Anti-Patterns to Capture Vulnerabilities knowledge

For a software engineer, a design pattern generally describes a good practice, and in turn, an anti-pattern, poor practice. However, sometimes good development practices are ineffective and turn into poor development practices. The use of anti-patterns allows developers to recognise commonly occurring problems, which may result from a lack of knowledge, insufficient experience in solving a particular type of problem, or applying a correct pattern in the wrong context (Foote and Yoder 1997, Dias e Silva 2014). Similarly, design patterns can turn into anti-patterns. The result is that a pattern that may be commonly used and generally considered good practice but is now ineffective and counterproductive in practice.

For example, common anti-patterns in critical or legacy systems are called *KEEPING IT WORKING* (Foote and Yoder 1997, Foote, Rohnert and Harrison 1999), which means do what it takes to maintain the software and keep it going and working. This anti-pattern is common in critical systems because when an essential element is broken, or a single failure will affect the entire system. An example is a system developed in C/C++ that uses unsafe function calls.

b) A Vulnerability Anti-Pattern

This is a final stage in which vulnerability knowledge is pushed into an anti-pattern structure to capture poor security practices. This leads to the derivation of a Vulnerability Anti-Pattern, a hybrid solution against prevalent vulnerabilities. This research proposed a VAP that captures cybersecurity domain knowledge and provides distilled knowledge access to enhance developer understanding so that software will not be exploited due to recurring vulnerabilities. Furthermore, the VAP offers precise information that extracted from specified knowledge sources such as vulnerability Databases (VDBs), Security Pattern (SP), Software Fault Pattern (SFP) and Attack Pattern (AP) to fulfil the need to provide essential vulnerability information.

a. General Structure of Vulnerability Anti-Pattern (VAP)

This section introduces the VAP general structure, which is comprised of

- Anti-pattern
- Pattern

VAP general structure explained further by VAP types in the following chapter.

a) Vulnerability Anti-Pattern General info

Anti-Pattern

a. Anti-Pattern Name:

b. Also Known as:

c. Most Frequent Scale in SDLC:

i. Requirement Specification Phase

ii. Design Phase

iii. Implementation/Coding Phase

d. Problem Description:

e. CWE Mapping:

i. CWE-ID

ii. General name

f. Related CWEs:

CWE-ID	Name

g. CVE Example:

b) Anti-Pattern Problematic Solution

a. Refactored Solution Name:

b. Refactored Solution Type:

i. Software Pattern

ii. Technology Pattern

iii. Process Pattern

iv. Role Pattern

c. Root Causes(Context):

d. Unbalanced Forces

Unbalanced forces	Attack	Example(code)
i. Management of functionality: meeting the requirements.		
ii. Management of performance: meeting the required speed of operation.		
iii. Management of complexity: defining abstractions.		
iv. Management of change: controlling the evolution of software.		
v. Management of IT resources: controlling the use and implementation of people and It artefacts		
vi. Management of technology transfer: controlling technology change.		

Pattern

e. Risk patterns and Consequences:

Risk Patterns	Consequences	Context Description
	STRIDE threat model	

f. Typical Causes

c) Problem Fingerprints(SFP)

i. Software Fault Pattern

d) Known Exploitation (Attack patterns-CAPEC)

i. Attack Pattern

e) Mitigation (Refactors the problem)

ii. Refactored Solutions:

i. Solution Steps

SDLC Phase	Solution

a. Examples: (Real world Patch example)

Product versions	Comment	Vulnerability	CVE-ID	Patch(solution)

b. Related Solutions (SP):

i. General Solution (All in one solution)

3.4.1.3 Knowledge Awareness Process (KAP)

Knowledge Awareness Process aims to provide vulnerability information to software developers through Vulnerability Anti-Patterns. The Notion of Vulnerability Anti-Pattern is a hybrid solution that intended to provide the developers with awareness of security flaws, so they will understand the vulnerabilities and their root-causes to deploy mitigation solutions. To achieve this, KAP is designed to measure the effectiveness and usability of VAPs for developers in identifying and understanding how malicious hackers can exploit vulnerabilities. The KAP is explained in detail in the evaluation Chapters 6, 7 and 8 of this thesis.

3.5 Conclusion

This chapter presented a critique of existing pattern-based approaches to security and suggested the formulation of a Vulnerability Anti-Pattern. The motivation of the Vulnerability Anti-Pattern approach was to create a pattern-based technique that resolves the problem of:

- Ineffectiveness of existing pattern-based approaches to providing information about the most commonly-occurring vulnerabilities. Software developers can use these to learn how a malicious hacker can exploit their software systems.
- A distinct knowledge gap between software engineers and cybersecurity experts in terms of how to create secure software systems.

The first problem requires a novel approach to capture vulnerabilities, one that is easily understood by developers.

The second problem requires a form of management that provides essential information flow of vulnerabilities to bridge the knowledge gap to help developers to create secure software. The VAP detailed design and created catalogues are presented in the next chapters, and then the performed evaluation is reported in subsequent Chapters 6, 7 and 8.

5 Creating a Catalogue of Vulnerability Anti-Patterns (VAPs)

This chapter presents a catalogue of Vulnerability Anti-Patterns (VAPs) that currently captures 12 vulnerabilities. However, each VAP has two types: formal and informal, so the catalogue includes 24 VAPs defending against 12 vulnerabilities. The included vulnerabilities in the catalogue chosen from OWASP Top 25 software errors list (Martin et al. 2011). These consist of the most critical security problems faced by today's developers. VAPs are clustered together by various classification schemes such as language-based or aggregation-based cluster.

The way VAPs are organised helps to explain to developers which VAP can be used against a particular type of vulnerability, coupled with prevention in the future. This chapter describes how the VAP catalogue was derived, and how the VAPs are organised in the catalogue.

This chapter's key objectives are:

- Sections 5.1 and 5.2 detail and explain the organisation of Vulnerability Anti-Patterns to develop a catalogue.
- Section 5.3 explains the organisation of VAPs.
- Section 5.4 describes the formal Vulnerability Anti-Pattern catalogue.
- Informal Vulnerability Anti-Pattern catalogue describes in Section 5.5.

5.1 Developing the Catalogue

The catalogue was created while considering evaluation studies of this thesis, in which vulnerabilities have been selected on the basis of participants' knowledge of the particular programming language and platform dependencies, so the evaluation study can assess the selected vulnerabilities awareness in participants and provide intervention through its related VAPs.

Catalogue of VAPs is comprised of most serious development errors identified by surveying various vulnerability databases and their recent trend reports. Then this research explored the root causes of development errors by surveying the multiple pattern-based approaches in order to deal with security such as security patterns, software fault patterns, and attack patterns. This was aided by the fact that this research proposed a new approach based on anti-pattern to capture software vulnerabilities as mentioned in Chapter 3, Section 3.4.1. The catalogue contributes to the organisation of all developed Vulnerability Anti-Patterns and its software errors.

There are multiple approaches to developing a catalogue, the most rigorous way to build a catalogue of VAPs is to enumerate every possible development error and their refactored solutions, build tools to implement candidate solutions as optimal answers, allow developers to use the tools and identify which refactored solutions are more useful, and finally include them in the catalogue. It is impossible for a researcher to follow this rigorous path; building tools and user testing each vulnerability (software error) in the catalogue would take hundreds of person years.

The proposed approach to develop a catalogue is mainly focused on fulfilling the recurrent vulnerabilities awareness need for software developers. However, instead of finding all possible vulnerabilities that occur due to poor coding practices, this research restricted to the most important 12 vulnerabilities, identified by the OWASP as “Most Dangerous Development Errors”. Furthermore, this research explores the most optimal and usable security solutions suggested by both communities (cybersecurity and software engineering). The catalogue of VAP is a summation of selected poor security coding practices and refactored solutions against vulnerabilities; it covers the solution space better than any arbitrary approach.

5.2 Vulnerability Anti-Patterns Clustering

5.2.1 Language-Based Cluster

In Figure 22, Language-based cluster represents all included vulnerabilities that are inherited in developers’ common devolvement practices not only due to a lack of poor understanding of security core concepts such as proper memory allocation and bound checking but important platforms, such as C/C++, also shared in the responsibility of the vulnerabilities’ reoccurrence. The motivation to put together this group of vulnerabilities into clusters are:

- All vulnerabilities occur due to specific languages such as C/C++ or PHP.
- All vulnerabilities have their origins in a lack of input validation or improper input validation checks.
- All vulnerabilities are mitigated to some extent by implementing/ following the secure coding practice.
- All vulnerabilities are led to information breaches such as denial-of-service (DoS) attack.

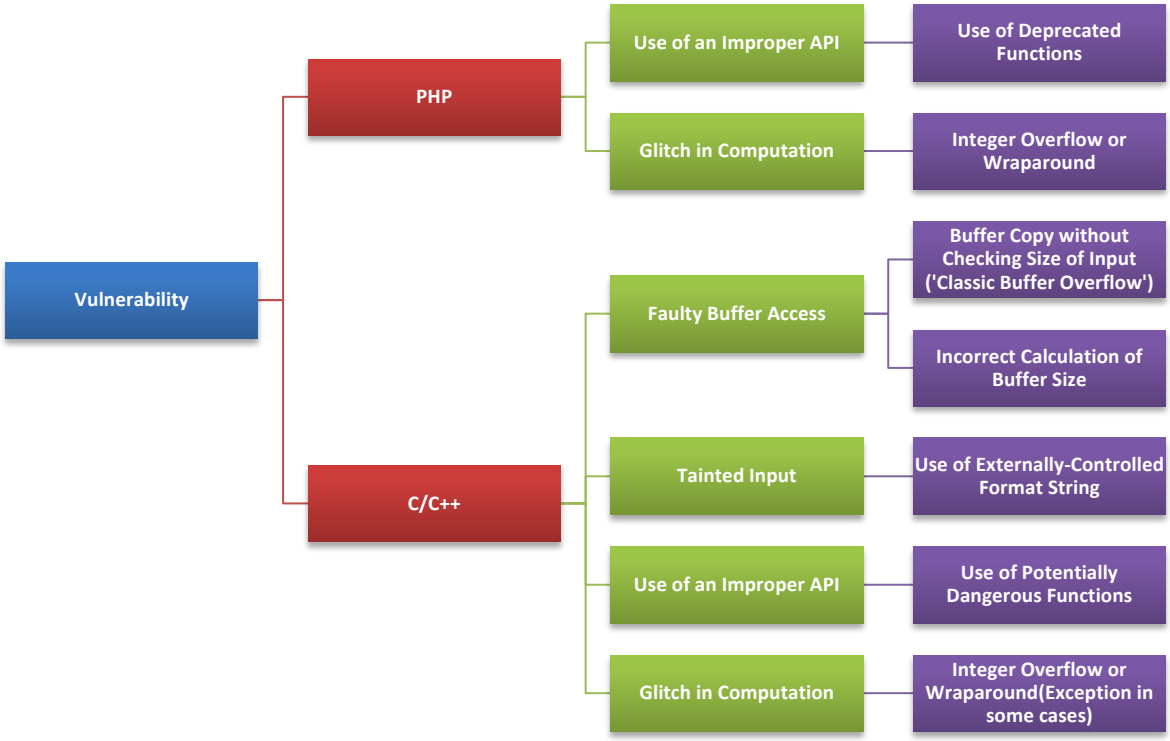


Figure 22 Language-based cluster

As shown in Figure 22, the language-based cluster appears to have two main languages:

- PHP
- C/C++

The leaf nodes of the cluster define the vulnerabilities and the middle nodes explain the lack of input validation or improper input validation checks that have been taken from the software fault pattern. Therefore, faulty computation leads to vulnerability.

5.2.2 Aggregation-Based Cluster Organisation

The cluster shown in Figure 23 is based on aggregation, which is a specialised form of association among vulnerabilities in order to indicate a significant correlation among vulnerabilities' root-causes. Fundamentally, "*Use of Potential Dangerous function*" is a parent vulnerability and "*Integer Overflow*", "*Buffer Overflow*", "*Incorrect Calculation of Buffer Size*", "*Use of Externally-Controlled Format String*" are child objects (vulnerabilities). This association among vulnerabilities help us to signify a hierarchical relationship as one of simplicity.

- **Parent Vulnerability:** Use of Potentially Dangerous Function
- **Child Vulnerabilities:** Integer Overflow, Incorrect Calculation of Buffer Size, Use of Externally Controlled Format String, Buffer Overflow.

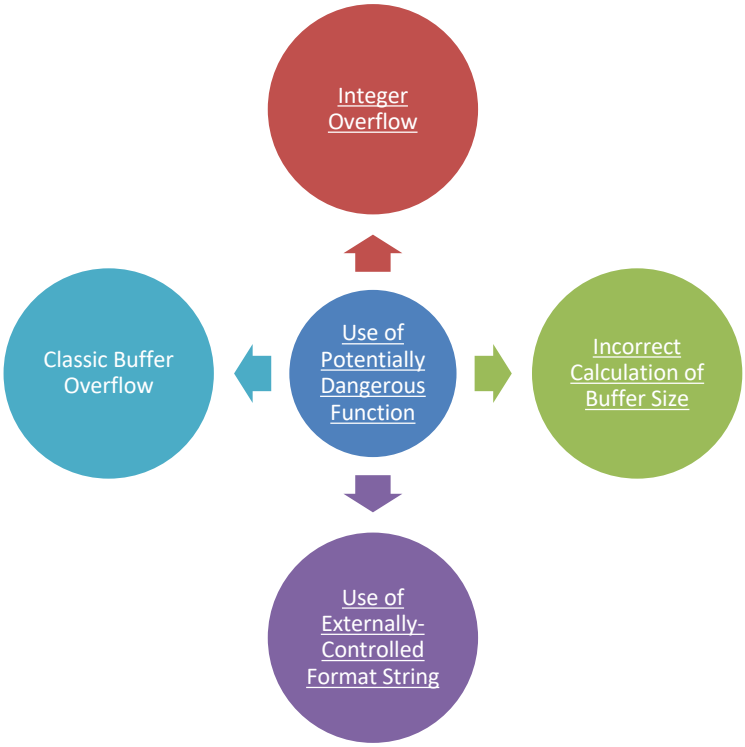


Figure 23 Aggregation-based cluster to display root causes with linkage parent and child vulnerabilities.

5.3 Organising Vulnerability Anti-patterns

Vulnerability Anti-Pattern catalogue organised into formal or informal types and then follows the language-based clustering as shown in Figure 24.

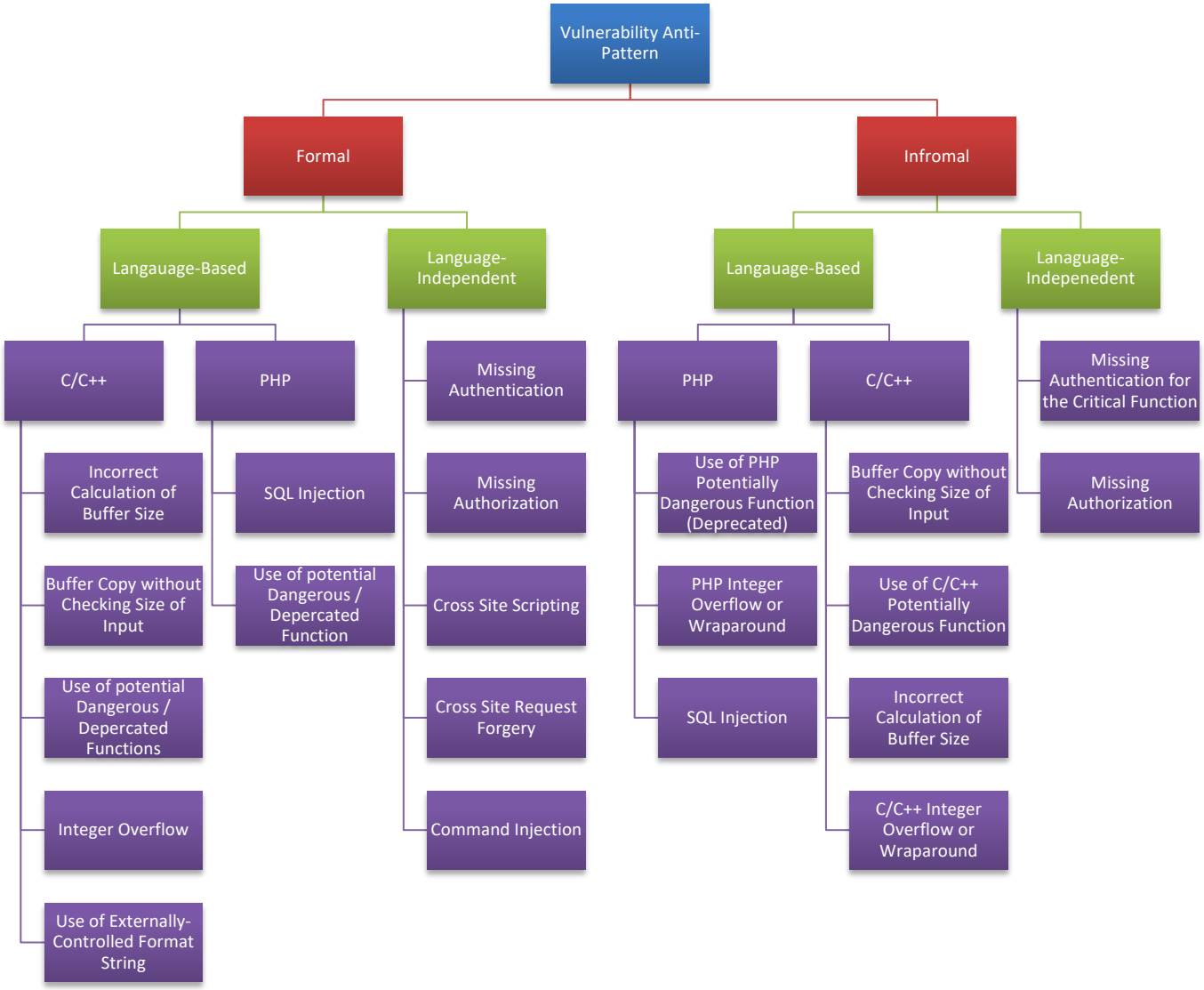


Figure 24 Organisation of Vulnerability Anti-Pattern

5.4 Vulnerability Anti-Pattern Catalogue

Figure 25 demonstrates the catalogue of included vulnerabilities with their sub-division into formal and informal arrangements.

1. **Formal Vulnerability Anti-Pattern Catalogue:**

As mentioned above in Section 4.6.2, the formal Vulnerability Anti-Pattern follows the standard format of anti-pattern to capture and present vulnerability.

2. **Informal Vulnerability Anti-Pattern Catalogue:**

As mentioned in Section 4.6.1, the Informal Vulnerability Anti-Pattern exemplifies the way in which a vulnerability can occur in a specific context.

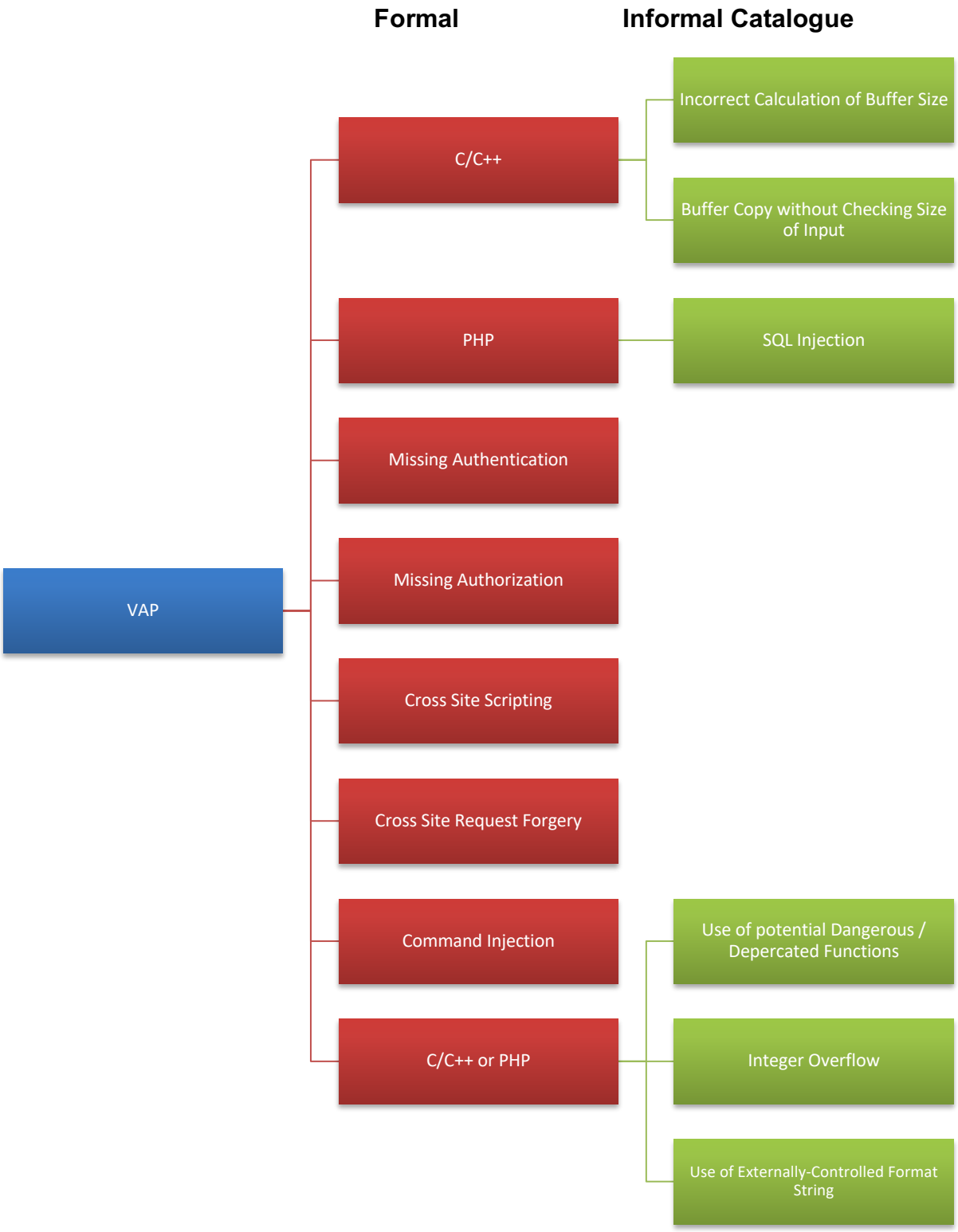


Figure 25 Hierarchical view of VAP catalogue

5.5 Informal Vulnerability Anti-Pattern Catalogue

As explained in Section 4.6.1, Informal vulnerability Anti-Pattern is exemplified the exploitation prospect for developers in an understandable format. The informal catalogue of Vulnerability Anti-Patterns includes 9 vulnerabilities:

1) Buffer Copy without Checking Size of Input
2) Use of C/C++ Potentially Dangerous Function
3) Use of PHP Potentially Dangerous Function (Deprecated)
4) Incorrect Calculation of Buffer Size
5) C/C++ Integer Overflow or Wraparound
6) PHP Integer Overflow or Wraparound
7) Missing Authentication for the Critical Function
8) Missing Authorization
9) SQL Injection

Details of each VAP explains in the appendix Section 1.2, which elucidates the relationship between vulnerability and its exploitation behaviour in the form of a pattern and anti-pattern.

5.6 Formal Vulnerability Anti-Pattern Catalogue

As explained previously in Section 4.6.2, formal vulnerability Anti-Pattern is extended explanatory configuration which elucidates detailed information of vulnerability for developers in an understandable format. The formal catalogue of Vulnerability Anti-Patterns includes 12 vulnerabilities:

1) Buffer Copy without Checking Size of Input
2) Use of Potentially Dangerous/ Deprecated Function
3) Incorrect Calculation of Buffer Size
4) Integer Overflow or Wraparound
5) Missing Authentication
6) Missing Authorization
7) SQL Injection
8) Improper Neutralization of Input during Web Page Generation
9) Cross-Site Request Forgery
10) Use of Externally-Controlled Format String
11) Shell Injection

Details of each pattern are explained in the appendix Section 1.1 that elucidates the factors involved to cause the vulnerability (i.e. anti-pattern and pattern).

5.7 Conclusion

The catalogue of Vulnerability Anti-Patterns (VAPs) has been created to match 12 vulnerabilities. VAPs have different types: formal and informal, which further is sub-categorised based on programming languages, such as PHP and C/C++. The included vulnerabilities in the catalogue have been chosen from the OWASP Top 25 software errors list (Martin et al. 2011) that consist of the most important security problems faced by today's developers. The way VAPs are organised helps explain to developers which VAPs can be used against a particular type of vulnerability coupled with prevention in the future. This chapter describes the VAP catalogue by various classification schemes such as language-based or aggregation-based clusters. The chapter concluded by presenting informal and formal VAP catalogues.

Evaluation

This section of the dissertation consists of four chapters to detail the results of three experimental studies: **Pilot study-I (PS-I),** which was conducted to test the proposed experiment design with Computing and Ethical hacking students at Abertay University, **Pilot study-II (PS-II),** which was performed only with Computing related degree students at Abertay University and an **Industrial study** performed with professional software engineers at a UK-based leading software development company. The evaluation is divided into three parts.

Chapter-6 Pilot study-I: the key objective was to evaluate developers' (in this case, students') understanding of security flaws that lead to vulnerabilities. We concluded this study by specifying outcomes which re-inform Vulnerability Anti-Patterns (VAPs) and the design of the subsequent pilot study-II and industrial study.

Chapter-7 Pilot study-II and **Chapter-8 Industrial study:** the key objectives were to evaluate developers' and students' understanding of security flaws that lead to vulnerabilities and to measure the effectiveness of VAPs to help developers in improving their understanding about vulnerabilities.

A mixed methods approach is used to evaluating developers' awareness about recurrent vulnerabilities and measuring the effectiveness of Vulnerability Anti-Pattern to provide understanding in a series of experiments as shown in Figure 26.

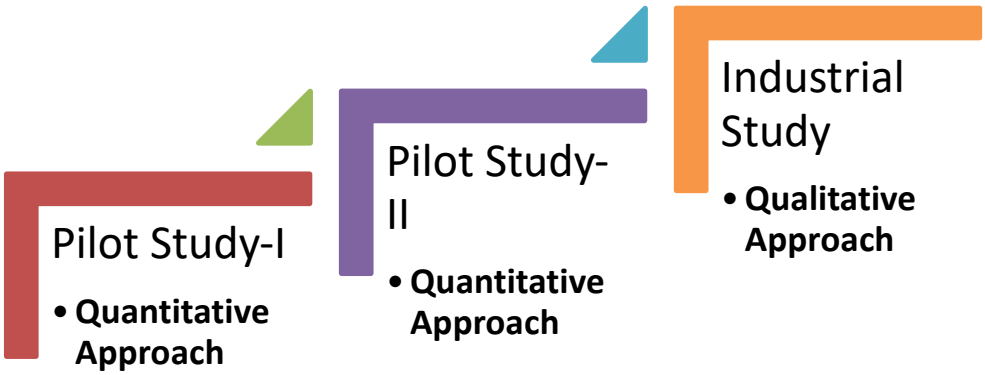


Figure 26 Experimental studies and their research methods approach

The derived pattern-based approach (VAP) mentioned in Chapter 4 is used as an intervention during **PS-II** and **Industrial study.**

Based on literature review analysis and Pilot-study-I results, it was apparent that the use of a pattern-based approach and subsequent development of VAP would be an appropriate solution. Therefore, an evaluation of VAPs was conducted via the **PS-II** and **Industrial study.**

Chapter-9 discusses all performed studies results and concludes by proving a brief summary of results to measure the effectiveness of the VAP for developers in order to provide essential awareness of vulnerabilities with support to the creation of secure software systems.

Table 20 describes the study questions (SQ) of pilot study-I, pilot study-II and industrial study. By analysing data in multiple ways, the results seek to determine:

Pilot Study-I	
SQ-1	Do software developers have an effective understanding of recurrent vulnerabilities?
Pilot Study-II & Industrial Study	
SQ-1	Do software developers have an effective understanding of recurrent vulnerabilities?
SQ-2	Can interventions based on Vulnerability Anti-Pattern help developers in improving their understanding of vulnerabilities?

Table 20 Study Objectives (SO) of the experimental studies.

The evaluation section of this dissertation is organised into 4 chapters: Chapter 6 presents the Pilot-study-I results and their statistical analysis; Chapter 7 presents the Pilot-study-II results, and their statistical analysis; Chapter 7 presents the Industrial study results and their qualitative analysis. Followed by Chapter 8 discusses all studies results.

6 Pilot Study-I (PS-I)

6.1 Introduction

Initially, a pilot study (PS-I) was conducted, to investigate the question that does software developers have an effective understanding of recurrent vulnerabilities? Furthermore, does PS-I designed questionnaire appropriately capture and present information about the selected vulnerabilities to participants in order to evaluate their understanding of vulnerabilities. In addition to this, there was a need to assess the complexity of the designed questionnaire for participants, the length of time required to complete the questionnaire and to identify any preliminary problems.

A small sample group of 30 computing degree related students were recruited as participants for PS-1. This study was performed to test the designed questionnaire appropriateness and selection of vulnerabilities. The PS-I participants were students of computer science majors at the Division of Computing and Mathematics, Division of Cybersecurity at Abertay University. The main issue reported by participants was the lengthy and time-consuming nature of the questionnaires as it was comprised of 10 vulnerabilities. To address these comments, the vulnerability sample size was reduced to 5.

The second pilot-study and the industrial study included only 5 vulnerabilities. The results of the pilot study are presented independently in Section 9.1.

6.1.1 General Description

This chapter reports the results of pilot study-I, which aimed to evaluate the developers' existing understanding of recurrent vulnerabilities. This study does not test an intervention based on Vulnerability Anti-pattern.

Furthermore, the PS-I analysed the results to determine the statistical significance to confirm the proposed hypothesis (as discussed in Section1.8). The results revealed that software developers lack an effective understanding of how to identify recurring security flaws (weaknesses) that enable the malicious attacker to carry out an attack.

6.1.2 Key Objectives

The Study Question (SQ) is described:

Do software developers have an effective understanding of recurrent vulnerabilities?

6.1.3 Experiment Hypothesis

The experiment study posits the following hypothesis (Experiment hypothesis=EH):

EH-1	Participants can identify recurring security flaws and know how malicious hackers can exploit these.
EH-0	Participants cannot identify recurring security flaws and do not know how malicious hackers can exploit these.

6.2 Method

6.2.1 Experiment Study Description

A pilot study was performed with students from **the School of Design and Informatics at the University of Abertay Dundee**. The study comprised a questionnaire, which investigated their awareness of the most commonly occurring software errors (i.e. vulnerabilities) and techniques for their mitigation. Computing related degree students such as BSc Ethical Hacking, BSc Computing, BSc Computer Games Technology, BSc Computer Application Development and MSc Ethical Hacking students were surveyed. All students share computer science as common knowledge background. However, ethical hacking degree students are different due to their major in cybersecurity, and they are also called penetration testers. Pilot study participants had alternative background knowledge. For example, ethical hacking students were expected to know some of the included vulnerabilities and their exploitation in comparison to computing or gaming degree students. The questionnaire consisted of 10 questions. Each question represented one particular vulnerability. The selected vulnerabilities were chosen from the "*2011 CWE/SANS Top 25 Most Dangerous*

Software Errors" list (Martin et al. 2011). 30 students participated and completed the questionnaire. Please see Appendices 1.2. For information. Each question was divided into three sub-parts,

Part-1: Investigated developers' (students) awareness of why a given piece of vulnerable code or UML diagram is insecure.

Part-2: Investigated developers' (students) ability to describe how this security flaw can enable a malicious attacker to carry out an attack.

Part-3: Investigated developers' (students) awareness of commonly used cybersecurity terminology to describe a security flaw.

In the questionnaire, each question has been assigned one of 4 points, which are distributed as follows:

- 0 = Student is unaware of the vulnerability.
- 1 = student is slightly aware of the vulnerability
- 2 = student is aware of the vulnerability
- 3 = student is well aware of the vulnerability

Each question sub-divided into 3 parts as shown in Table 21; each part is manually marked based on the answers of participants by researchers and moderated by someone else. The points assigned to the participants according to the above mentioned criteria (between 0 to 3). An example question is explained below:

<table>
<tr><td colspan="2">Use the C# code sample below to answer the following questions I, II and III?
1. string userName = ctx.getAuthenticatedUserName(); //function call to get user name
2. string query = "SELECT * FROM items WHERE owner = '" + userName + "' AND itemname = '" + ItemName.Text + "'"; // database query to retrieve item information using user name and itemname as an input value from user.
3. sda = new SqlDataAdapter(query, conn);// execute the query

The program output will retrieve item information that corresponds to the username and itemname.</td></tr>
<tr><td>Part-1</td><td>I. The above code contains a flaw, which may not be detected by the compiler. Please describe below why you think this code is wrong.</td></tr>
</table>

Part-2	II.	A malicious hacker could exploit this code to concatenate malicious input to build SQL command to skip any input validation on username and can be used to gain access to database information. Can you explain how this code would enable a malicious hacker to carry out such an attack?
Part-3	III.	Can you explain this security flaw?

Table 21 PS-I question example

6.2.2 Experiment Design Structure

The rationale for the experimental design displayed in Table 22, which comprised:

- **Assessment Survey Study**

Accessing and measuring participants' (software developers') awareness about prevalent vulnerabilities.

Stage-1	**Assessment Survey Study**	
	Input	Questionnaire with vulnerable code/ UML diagram
	Output	Assessed result of developers' awareness of vulnerabilities

Table 22 Pilot-Study-1 experiment design structure

6.2.3 Experiment Questions' Structure

The structure of each question was constructed on the basis of shared information accumulated from both communities' sources: software engineering and cybersecurity. Each question consisted of three parts:

Part-1: Vulnerable code or UML diagram

Part-2: Misused or exploited technique

Part-3: Identify the vulnerability's formal name (Formally defined by cybersecurity community)

6.2.4 Vulnerability Sample Size

This study considered the NVD (NIST 2011) as the main source of vulnerabilities' related data, which includes CVE as a sub-set repository to trace and track the most serious vulnerabilities. This research aims to structure information of those software errors that have serious and dangerous consequences for software systems. Table 23 presents the information on selected vulnerabilities from the list of "*2011 CWE/SANS Top 25 Most dangerous software errors*" (Martin et al. 2011) according to students' experience of a particular programming language. The experiment includes the following vulnerabilities:

Question Number	Vulnerability OWASP Rank	Vulnerability Name	Example Code Language
Q1	Rank 24	Integer Overflow	C/C++, PHP
Q2	Rank 18	Use of Dangerous Function Call	C/C++
Q3	Unranked	Integer to Buffer Overflow	C++
Q4	Rank 23	Use of Externally-Controlled Format String	C++
Q5	Rank 3	Buffer Overflow	C++
Q6	Rank 20	Incorrect Buffer Size Calculation	C++
Q7	Rank 18	Use of Dangerous Function Call	C++
Q8	Rank 1	SQL Injection	PHP
Q9	Rank 6	Missing Authorization	UML Class diagram
Q10	Rank 5	Missing Authentication	UML Class Diagram

Table 23 Vulnerabilities included in questionnaire

6.2.5 Participants' Sample Size

In this experimental study, the term "penetration tester" is used here to refer to ethical hacker and research students, and the term "software developers" will be used in its broadest sense to refer to all students of computing, computer games technology and computer games application development technology. Table 24 presents the participants' demographics.

Number of participants	Degree Title	Year of study
1	Computer Games Application Development	1st
1	Computer Games Application Development	3rd
6	Ethical Hacking	3rd
4	Computer Games Technology	1st
1	Computer Games Technology	2nd
1	Computer Games Technology	3rd
2	Computing	1st
7	Computing	3rd
3	Computing	4th
2	Post-Grad-Research	

Table 24 Participants information

All participants were considered to be software developers during PS-I. However, after their score analysis, the PS-I found significant trends in the participants, whose major were cybersecurity. To investigate significant outcome, they were classified in two groups.

1. **Penetration Tester**
    - Post-grad Research Students (all participants were from the cybersecurity division)
    - Ethical Hacking Students
2. **Software Developers**
    - Computer Games Technology Students
    - Computer Application Development Technology Students
    - Computing Students

To evaluate students' performance depending on their degree, each degree is assigned a code:

1=Computing

2=Ethical hacking

3=Computer Games Technology

4=Computer Game Application Technology

5=Post-Graduation

6.3 Results

6.3.1 Assessment of Questionnaire Vulnerabilities

This section presents the mean scores attained for each vulnerability assessed by the questionnaire. The result of each question is displayed as a bar chart to reflect on how participants' obtained marks and their related degrees.

6.3.1.1 Integer Overflow Vulnerability

When participants were asked about an integer overflow vulnerability during the experiments, Computer Game Application students got the highest marks. However, computing students got the lowest marks; the mean of their total obtained marks out of 3 is shown in Figure 27.

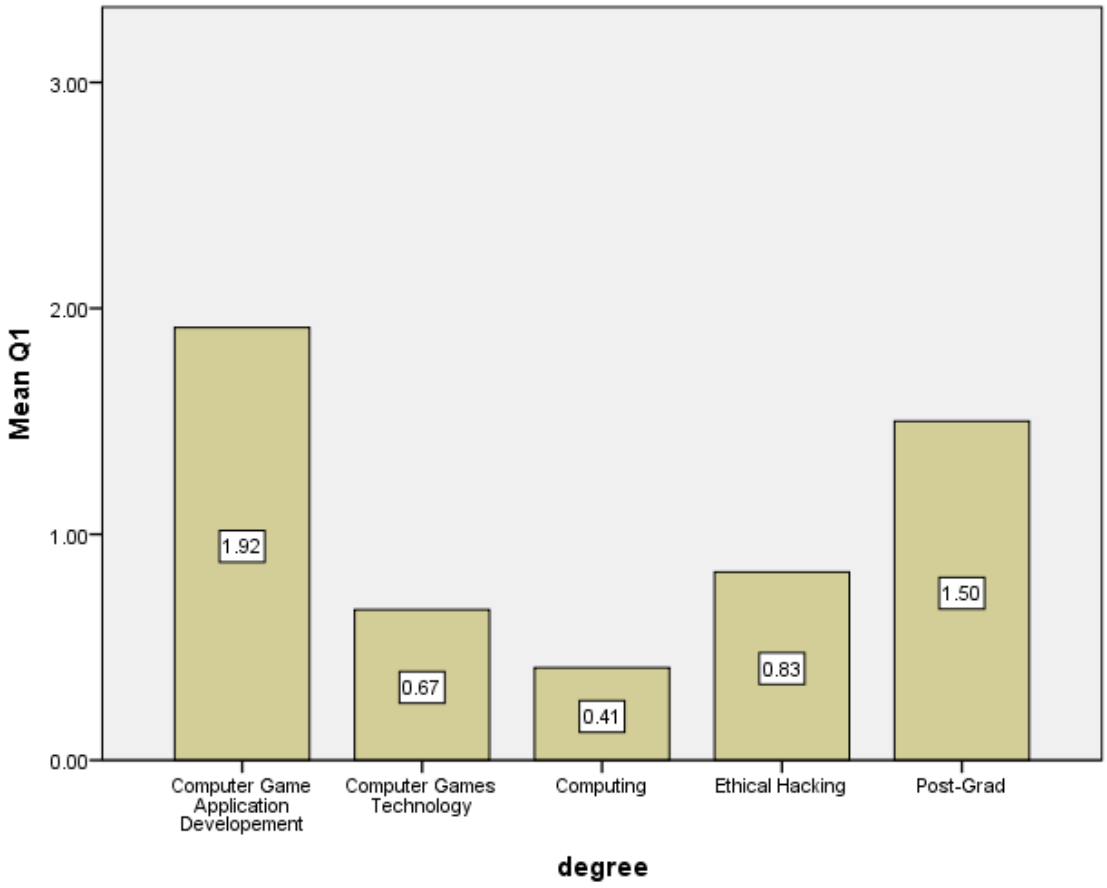


Figure 27 Integer overflow vulnerability mean score

6.3.1.2 Use of Dangerous Function Call in C++ Vulnerability

When participants were asked about the use of dangerous function calls during the experiments, Post-Grad students obtained the highest marks. However, computer game application students obtained the lowest marks; the mean of their total obtained marks out of 3 is shown in Figure 28.

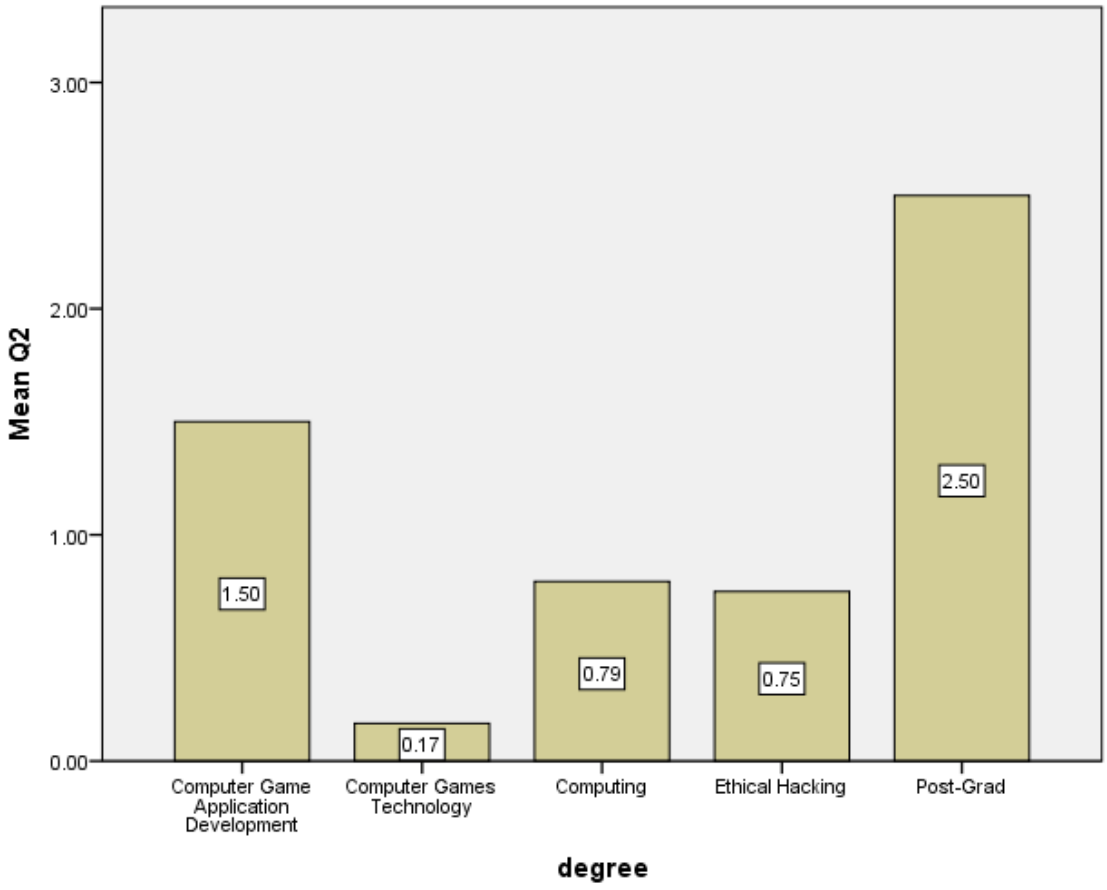


Figure 28 Dangerous function call in C++ vulnerability mean score

6.3.1.3 Integer to Buffer Overflow Vulnerability

When participants were asked about an integer to buffer overflow vulnerability during the experiments, the majority of participants struggled to find the vulnerability in the vulnerable code. However, post-Grad and ethical hacking students managed to answer correctly; the mean of their total obtained marks out of 3 is shown in Figure 29.

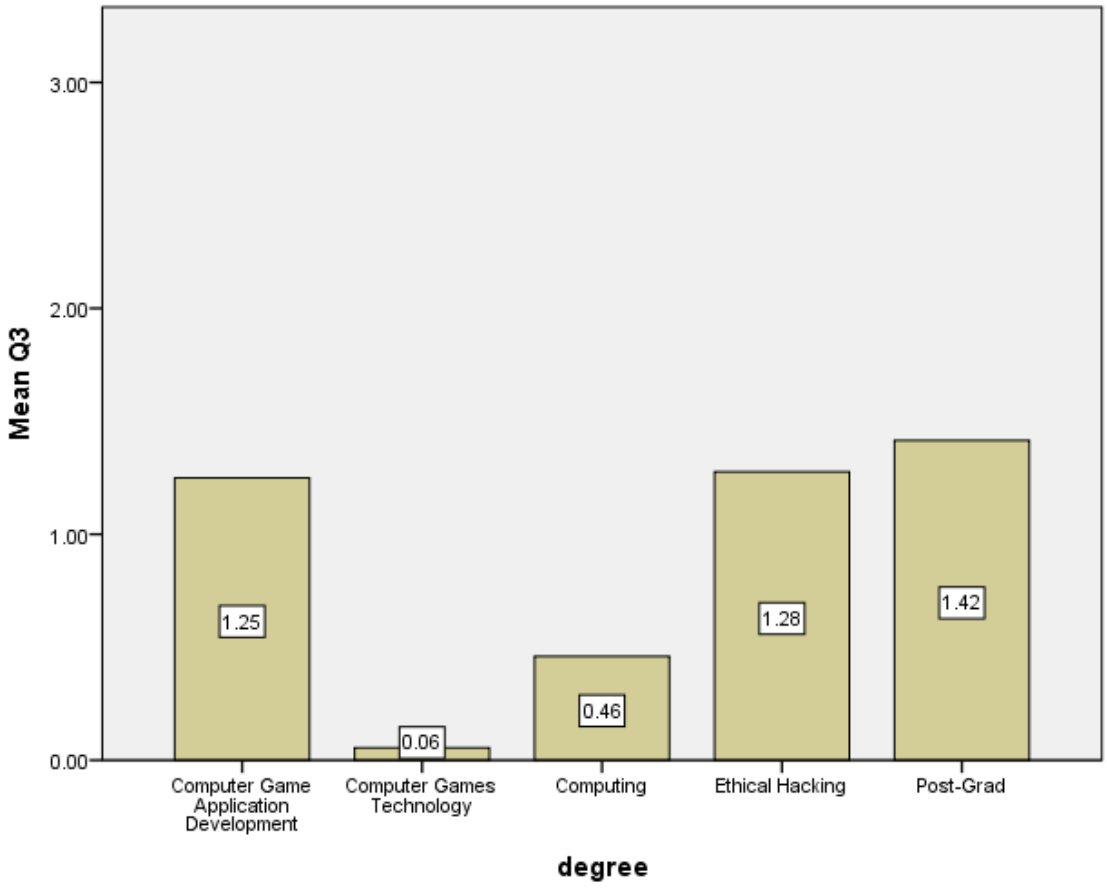


Figure 29 Integer to buffer overflow vulnerability mean score

6.3.1.4 Use of Externally-Controlled Format String Vulnerability

When participants were asked about the use of an externally-controlled format string vulnerability during the experiments, the majority of participants were not able to answer any questions relating to this vulnerability. However, some of the Post-Grad and ethical hacking students managed to answer correctly; the mean of their total obtained marks out of 3 is shown in Figure 30.

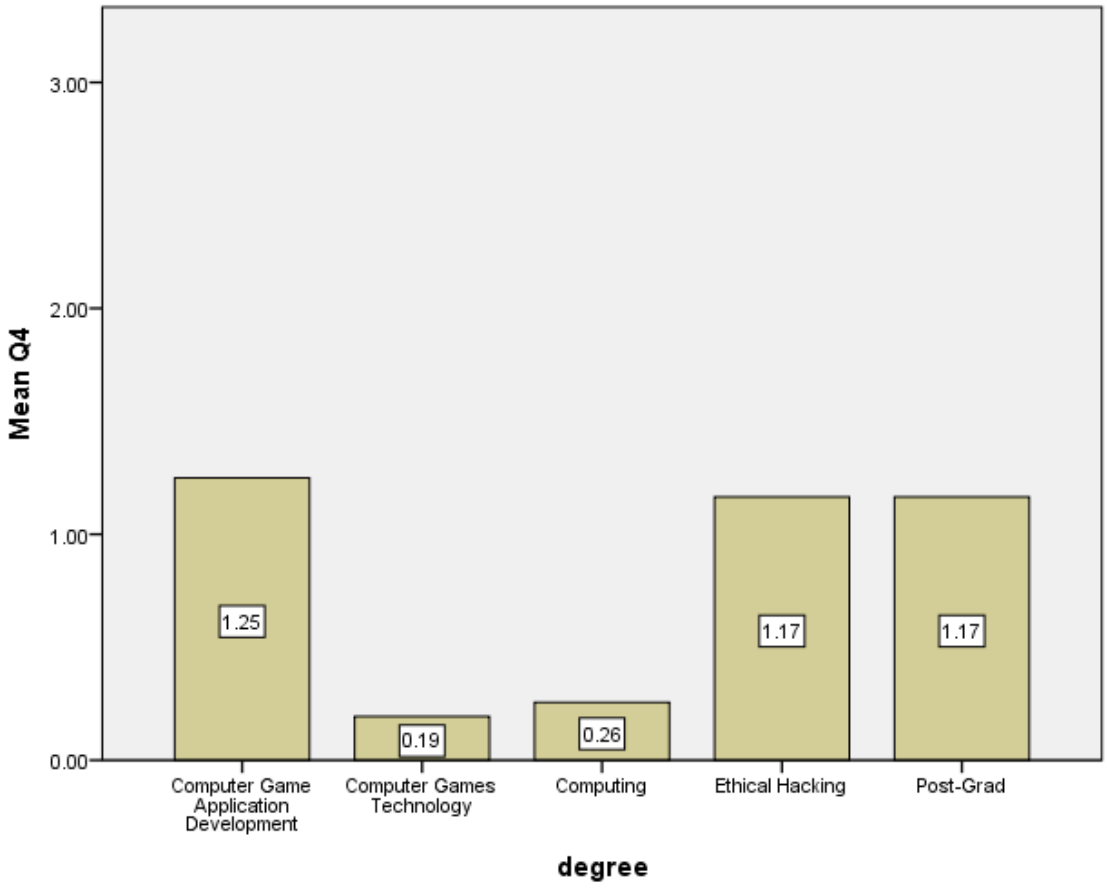


Figure 30 Use of externally-controlled format string vulnerability mean score

6.3.1.5 Buffer Overflow Vulnerability

When participants were asked about a buffer overflow vulnerability during the experiments, the buffer overflow vulnerability appeared difficult for computing and computer games technology students; the mean of their total obtained marks out of 3 is shown in Figure 31.

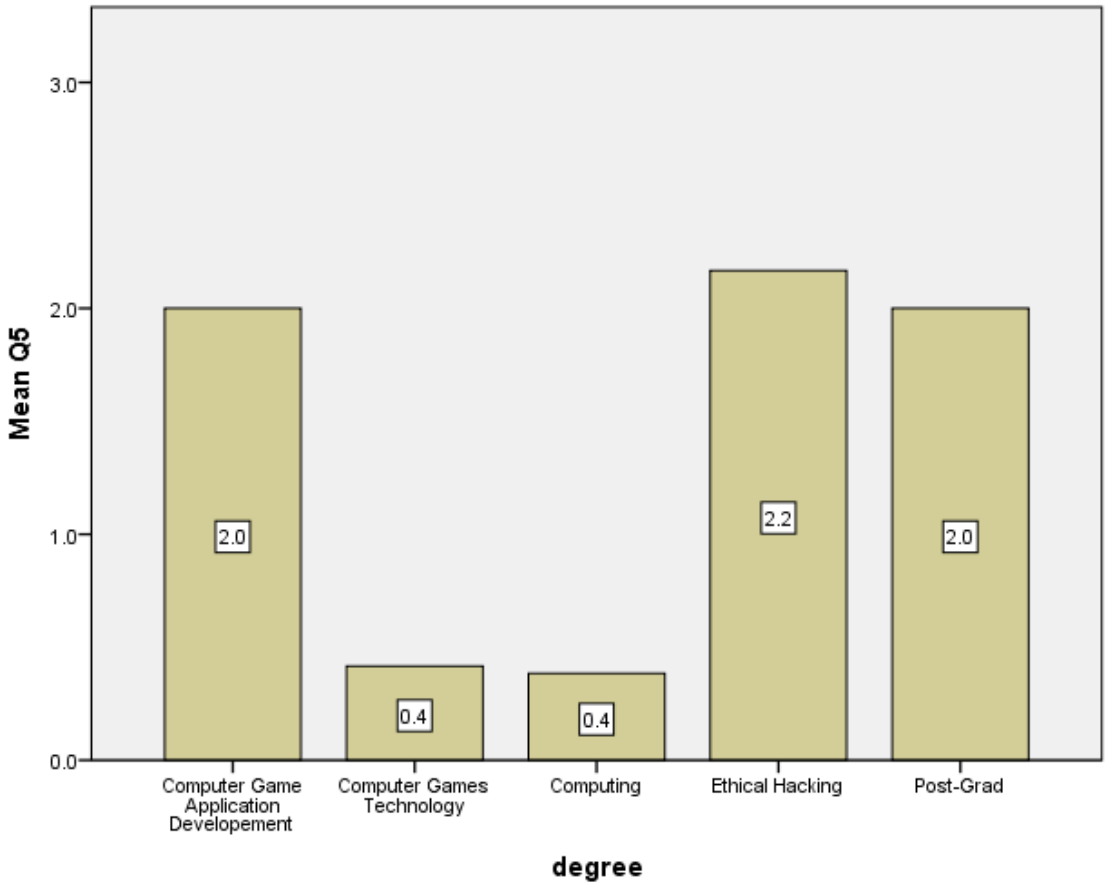


Figure 31 Buffer overflow vulnerability mean score

6.3.1.6 Incorrect Buffer Size Calculation Vulnerability

When participants were asked about an incorrect buffer size calculation vulnerability during the experiments, like the buffer overflow, this vulnerability appeared difficult for computing and computer games technology students; the mean of their total obtained marks out of 3 is shown in Figure 32.

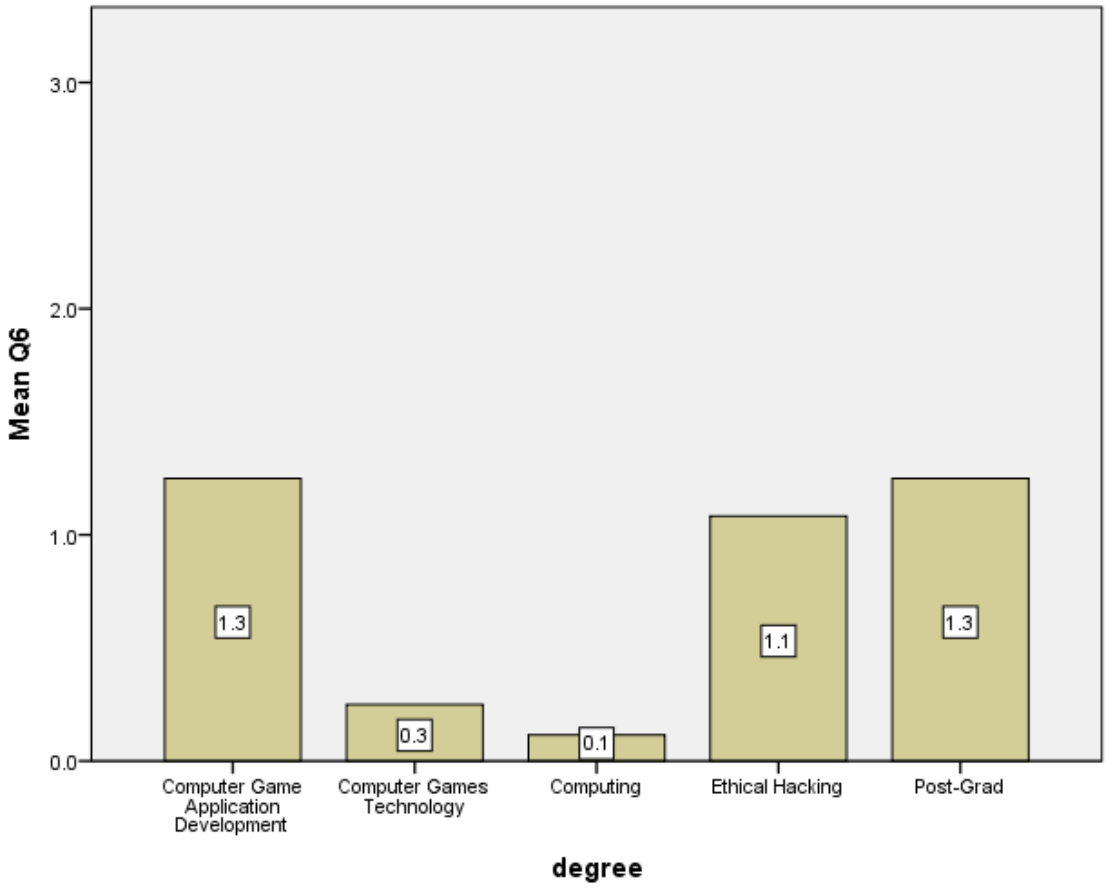


Figure 32 Incorrect buffer size calculation vulnerability mean score

6.3.1.7 Use of Dangerous Function Call in PHP Vulnerability

When participants were asked about the use of a dangerous function call in PHP vulnerability during the experiments, Post-Grad students got the highest scores; the mean of their total obtained marks out of 3 is shown in Figure 33.

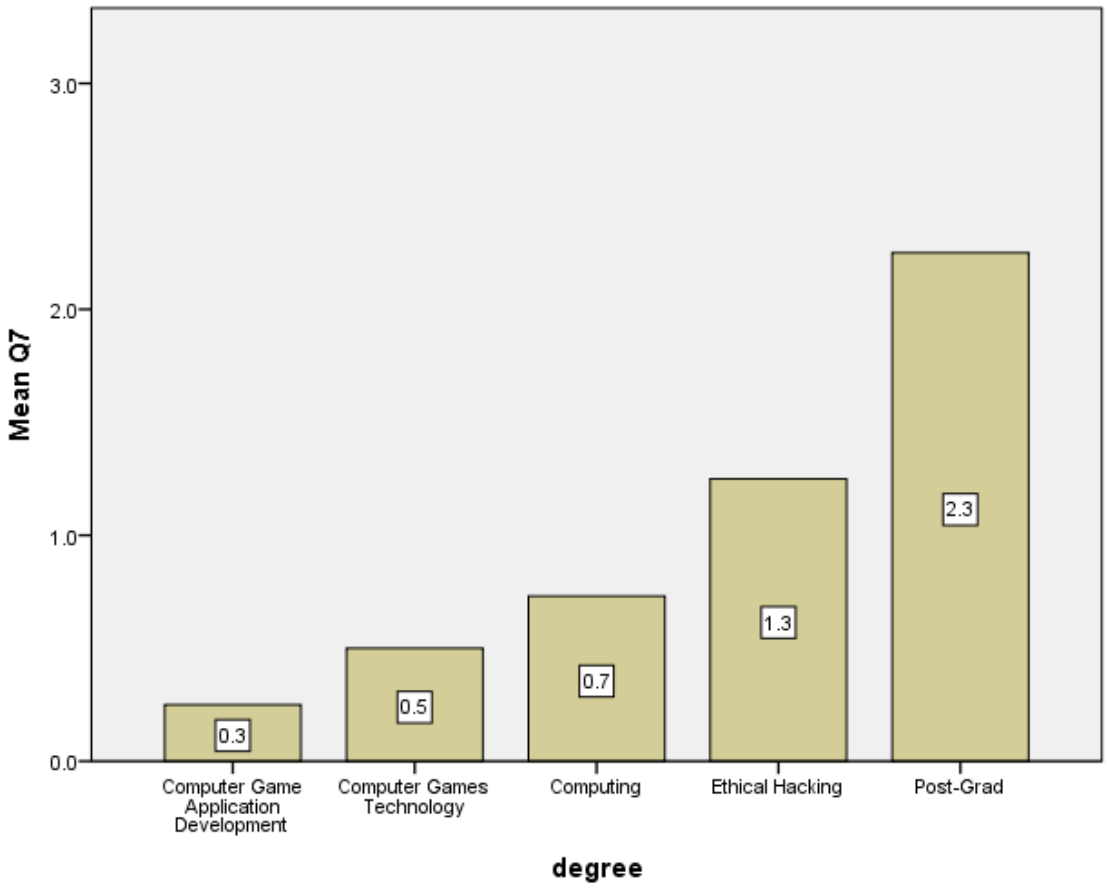


Figure 33 Use of Dangerous function call in PHP vulnerability mean score

6.3.1.8 SQL Injection Vulnerability

When participants were asked about an SQL Injection Vulnerability during the experiments, Post-Grad and ethical hacking students got the highest scores; the mean of their total obtained marks out of 3 is shown in Figure 34.

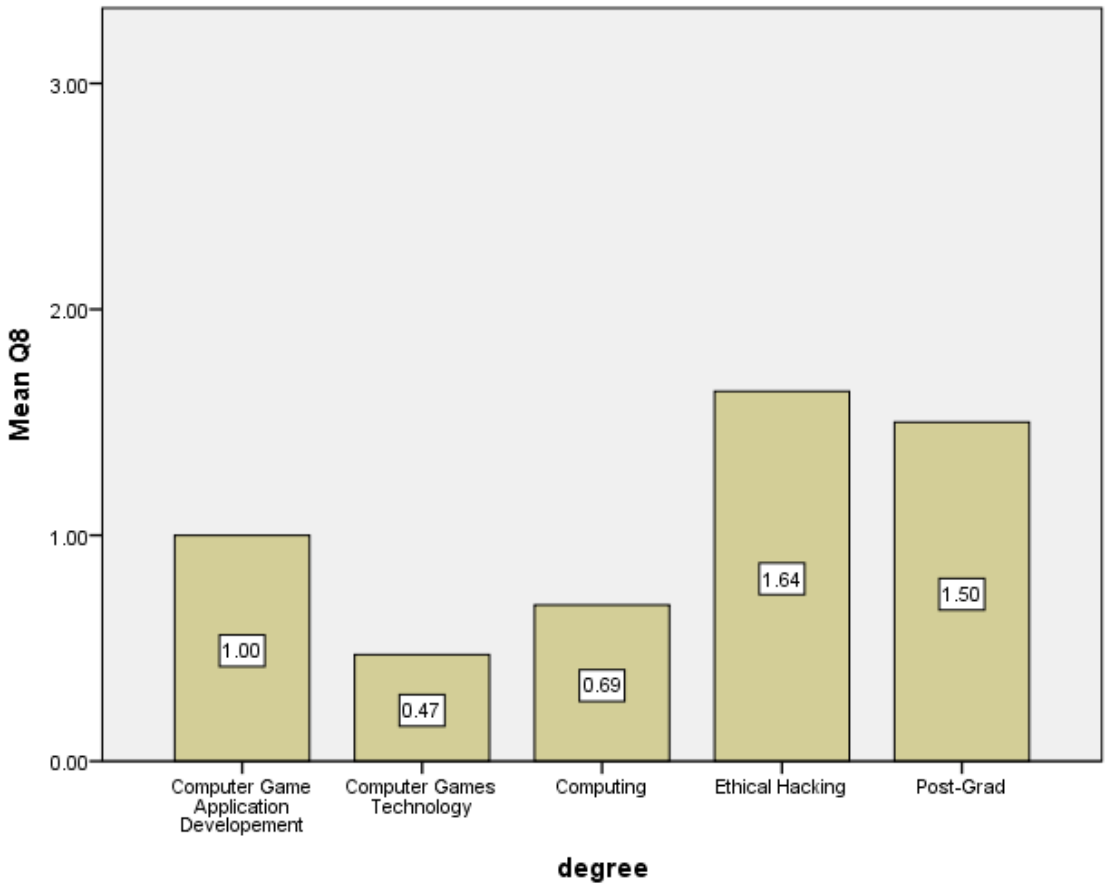


Figure 34 SQL injection vulnerability mean score

6.3.1.9 Missing Authorization Vulnerability

When participants were asked about a missing authorization vulnerability during the experiments, for computer game application development students this vulnerability was difficult to identify; the mean of their total obtained marks out of 3 is shown in Figure 35.

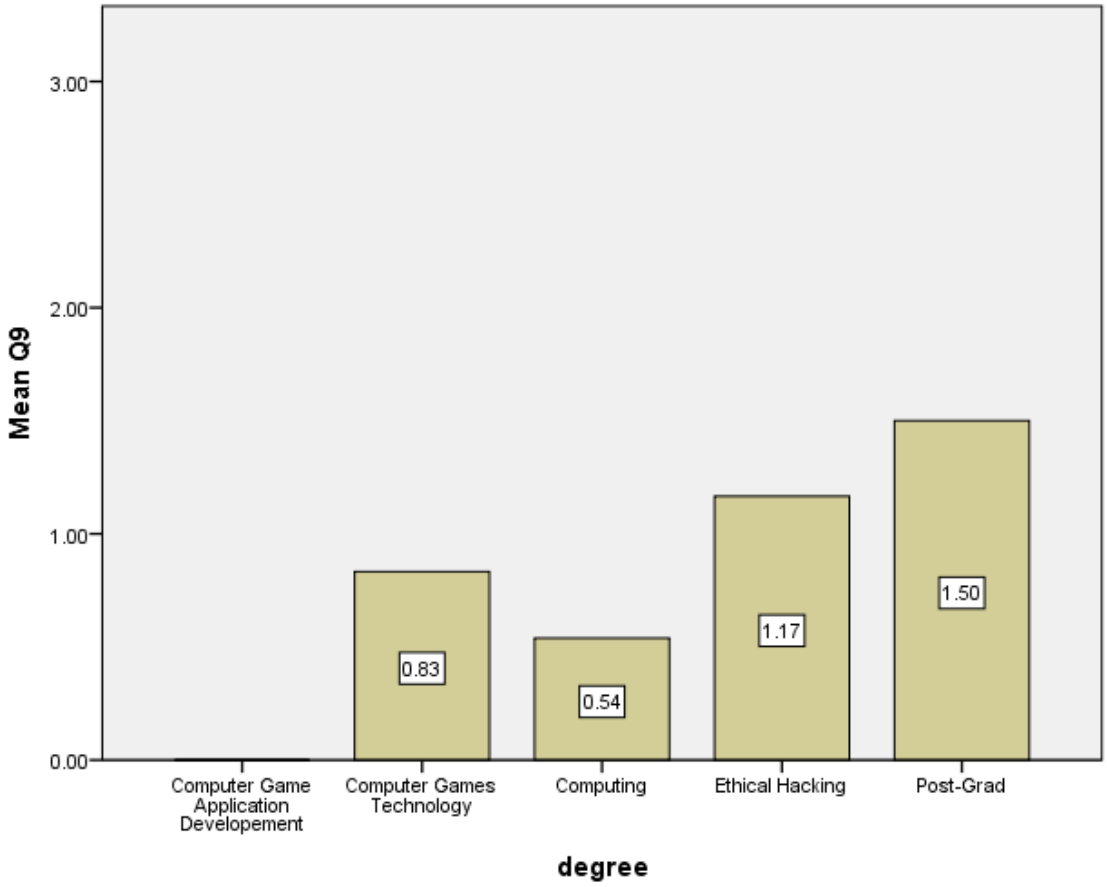


Figure 35 Missing authorization vulnerability mean score

6.3.1.10 Missing Authentication Vulnerability

When participants were asked about missing authentication vulnerability during the experiments, computing and computer games application students found it difficult to identify this vulnerability. The mean of their total obtained marks out of 3 is shown in Figure 36.

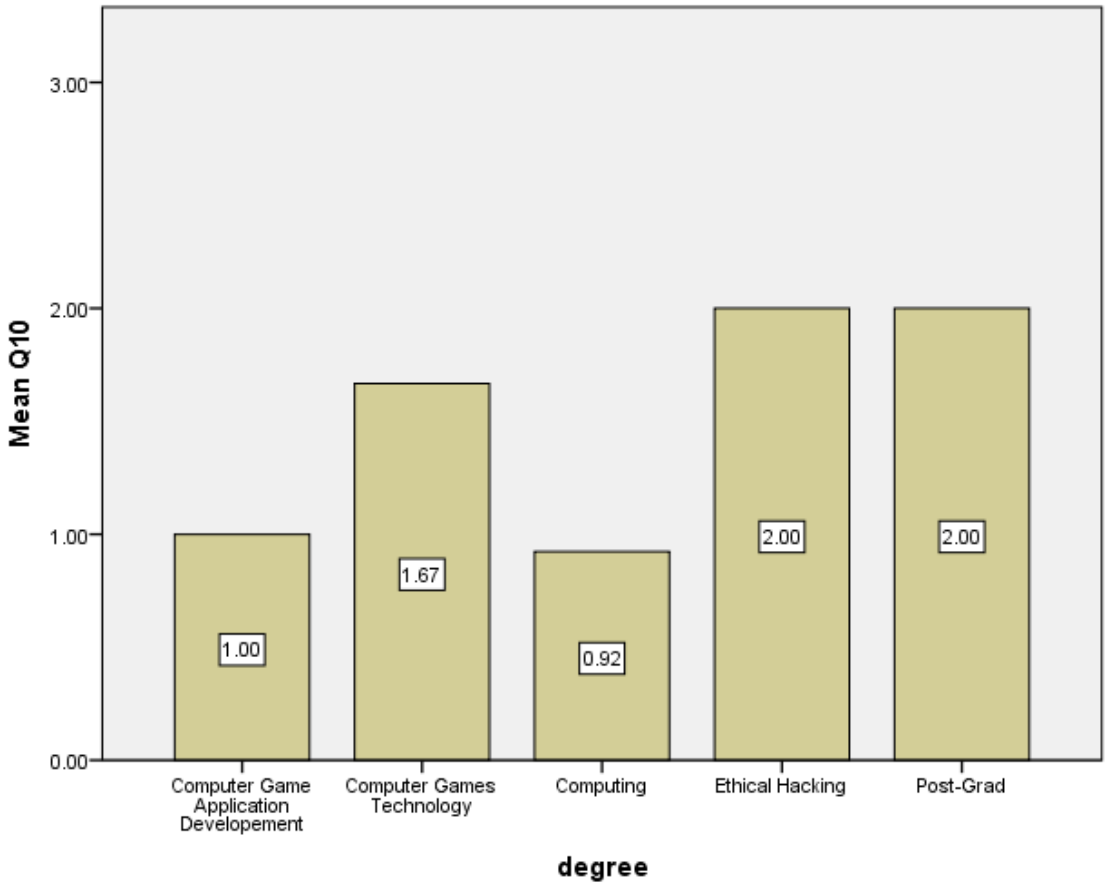


Figure 36 Missing authentication vulnerability mean score

6.3.1.11 Discussion

Students with a cybersecurity background performed well in identifying vulnerabilities. This group includes ethical hacking and post-grad students only however, computing-related degree student group performed well in demonstrating knowledge of vulnerabilities related to coding such as buffer overflow and incorrect buffer size calculation.

In PS-I, all participants were considered to be software developers, because they all studied computer science. However, their scores showed that penetration testing groups performed better in all questions.

6.3.2 Results Discussion for Total Scores of Vulnerabilities Questions

The Mean Total Score was performed to evaluate and compare all participants' scores with respect to their degree. Figure 37 graph x-axis represents the students' degree and the y-axis representing the mean total score obtained by participants. The score of ethical hacking and post-grad students appeared to be higher, although the score belonging to students of the Computing, Computer Games Technology and Computer Games Application Development degrees are lower indicating that penetration testers (Ethical Hacking degree students) are more efficient in finding vulnerabilities in vulnerable code samples relative to software developers. In other words, software developers received lower scores than penetration testers. It may be due to influences in their background knowledge and awareness of vulnerabilities. penetration testers and post-grad students studied ethical hacking modules that provide students with a deep understanding of vulnerabilities. This is in comparison to software developers, whose background knowledge is influenced by programming languages and advanced software development modules.

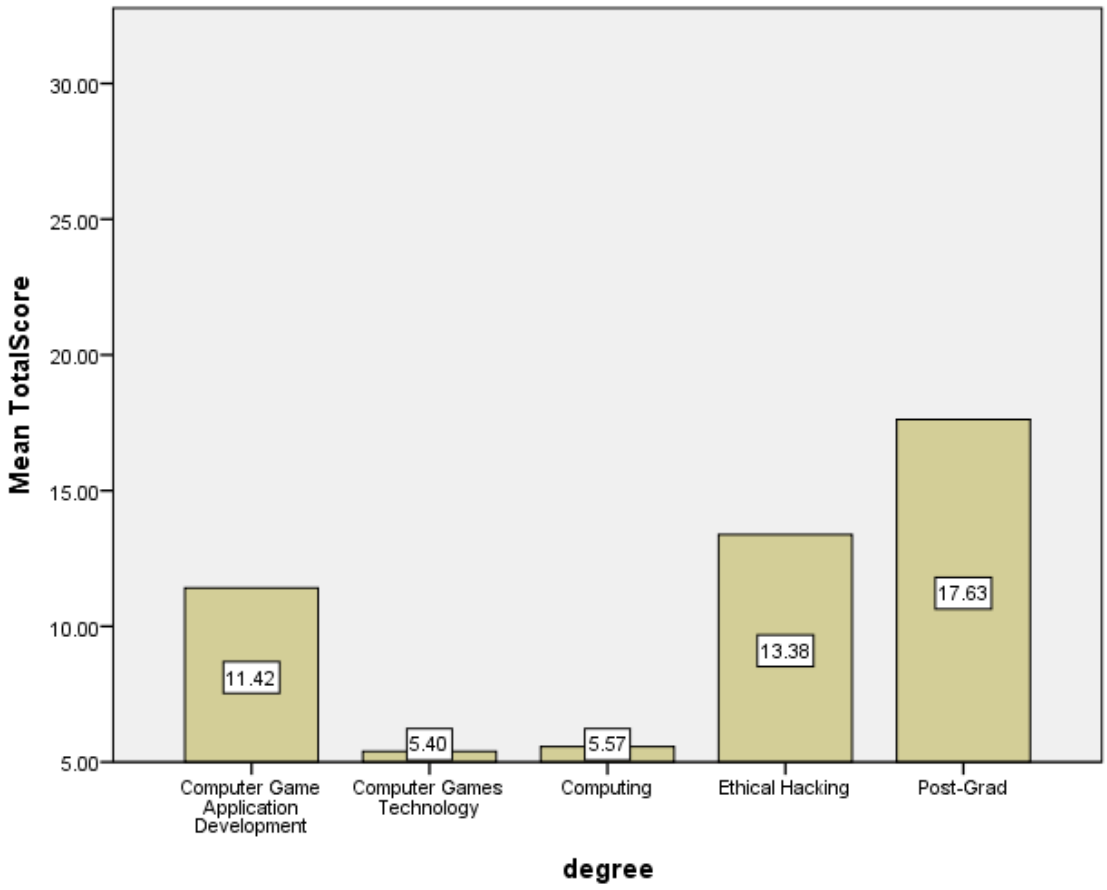


Figure 37 Total scores mean

6.3.3 Results Discussion of Mean of Total Score Graph

Figure 38 shows the accumulated results after combining all computing-related degree students as software developers and comparing to post-grad and ethical hacking degree students. The mean total score comparison of software developers and ethical hackers reveals that there was a sharp difference in ethical hackers scores, in comparison to software developers scores. Ethical hackers score 50% better than software developers in finding vulnerabilities.

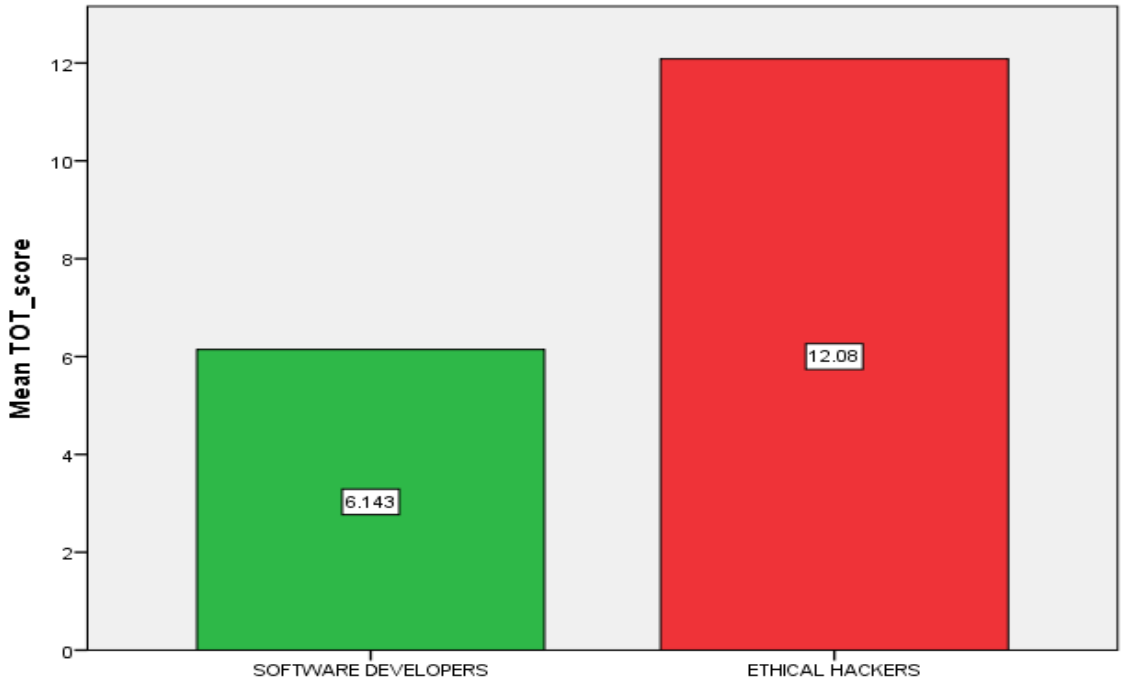


Figure 38 Total score mean comparison between software developers and pen tester

6.3.4 Assessment of Questionnaire Data-Statistical Analysis

A number of statistical methods were used to analyse the data gained from the questionnaires.

6.3.4.1 Checking for Normality Distribution

The author performed a four-step procedure to check if parametric assumptions are satisfied or not while using SPSS.

Step-1: Means and median are not similar between degree code 2 (Ethical hacking) and 3 (Computer Game Technology) as shown in Table 25.

Step-2: However, the means of all degree codes are higher than the standard deviation.

Table 25 illustrates that the degreecode 5 (post-graduate) students perform very well and degreecode 2 (ethical hacking) students perform second best among all other degrees' students, though, degreecode 1 (Computing) students perform least good.

Descriptives

	degree			Statistic	Std. Error
TotalScore	Computer	Mean		6.6013	1.57611
		95% Confidence Interval for Mean	Lower Bound	2.8743	
			Upper Bound	10.3282	
		5% Trimmed Mean		6.5108	
		Median		7.6650	
		Variance		19.873	
		Std. Deviation		4.45792	
		Minimum		.00	
		Maximum		14.83	
		Range		14.83	
		Interquartile Range		4.83	
		Skewness		.442	.752
		Kurtosis		.949	1.481
	Computing	Mean		5.5700	1.07363
		95% Confidence Interval for Mean	Lower Bound	3.2308	
			Upper Bound	7.9092	
		5% Trimmed Mean		5.5411	
		Median		5.4900	
		Variance		14.985	
		Std. Deviation		3.87104	
		Minimum		.00	
		Maximum		11.66	
		Range		11.66	
		Interquartile Range		6.75	
		Skewness		.192	.616
		Kurtosis		-1.062	1.191
	Ethical	Mean		13.3833	2.02251
		95% Confidence Interval for Mean	Lower Bound	8.1843	

			Upper Bound	18.5824	
		5% Trimmed Mean		13.4448	
		Median		14.9100	
		Variance		24.543	
		Std. Deviation		4.95412	
		Minimum		7.16	
		Maximum		18.50	
		Range		11.34	
		Interquartile Range		9.96	
		Skewness		-.507	.845
		Kurtosis		-2.093	1.741
	Post-Grad	Mean		17.6250	5.29500
		95% Confidence Interval for Mean	Lower Bound	-49.6544	
			Upper Bound	84.9044	
		5% Trimmed Mean		.	
		Median		17.6250	
		Variance		56.074	
		Std. Deviation		7.48826	
		Minimum		12.33	
		Maximum		22.92	
		Range		10.59	
		Interquartile Range		.	
		Skewness		.	.
		Kurtosis		.	.

Table 25 Descriptive analysis

Step-3: Test of Normality

The Tests of Normality as shown in Table 26, it is clear that the p-values 0.333, and 0.604 are higher than 0.05 so it can be assumed that the data is normally distributed.

Tests of Normality

degreecode		Develop_EH	Kolmogorov-Smirnov[a] Statistic	df	Sig.	Shapiro-Wilk Statistic	df	Sig.
1	TOT_score	1	.181	12	**.200***	.925	12	**.333**
2	TOT_score	2	.213	6	**.200***	.933	6	**.604**

Table 26 Tests of Normality

Step-4: Histogram

Figure 39 shows a histogram that was generated to explore the data. The graph is not bell-shaped, and data is not symmetrical around the mean.

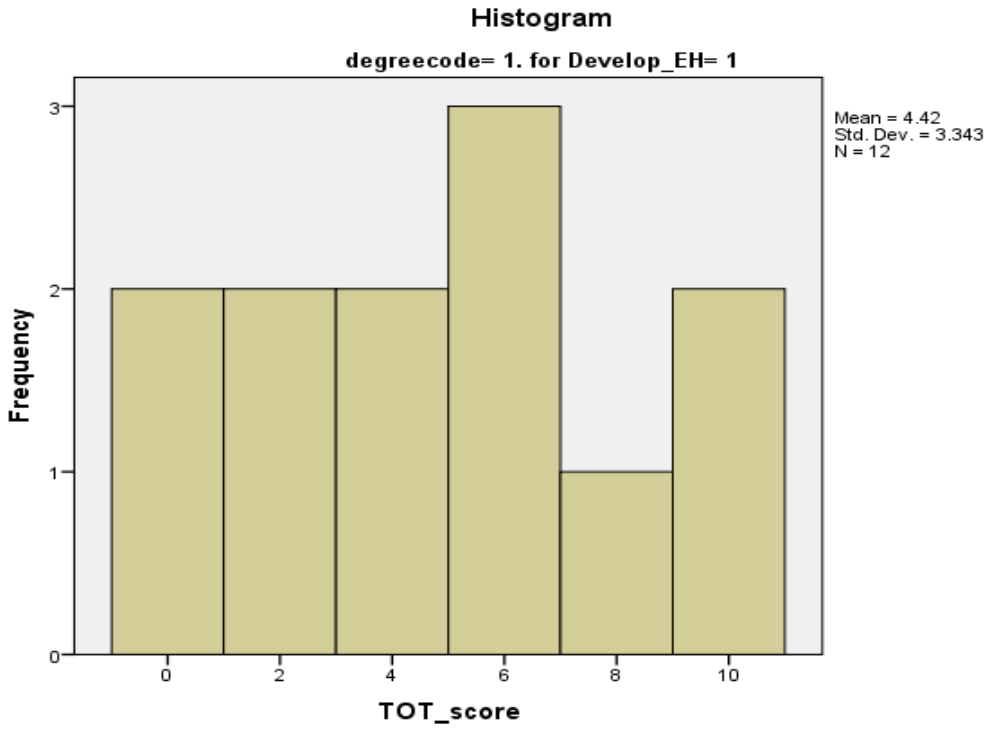


Figure 39 Histogram from the data explore output

Step 5: Normality Plots

Figure 40 shows that data is not normally distributed. There are some noticeable outliers at the top and bottom ends.

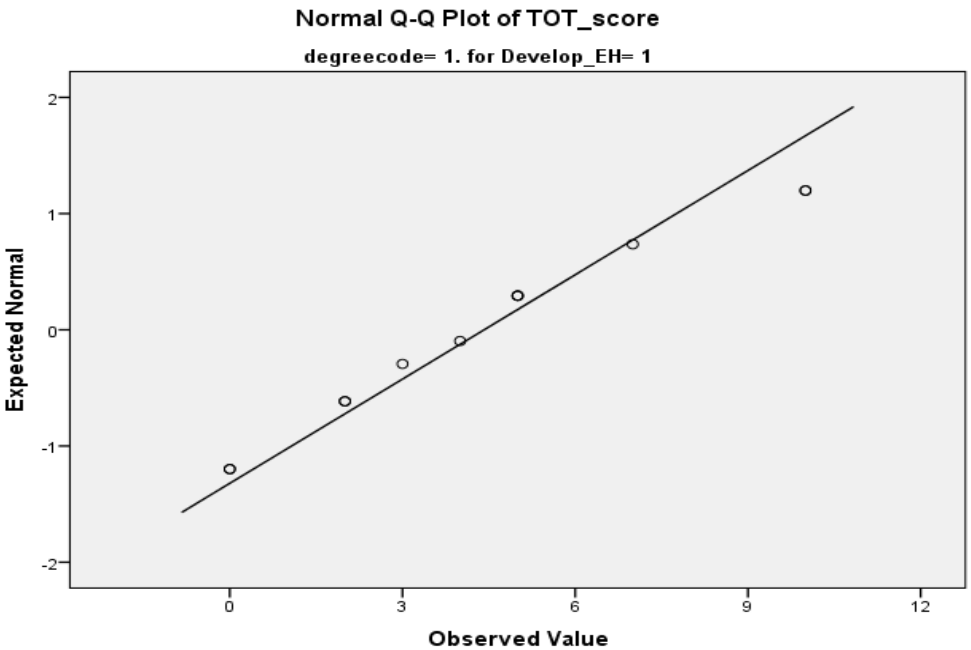


Figure 40 Normality plot from the data explore output

6.3.4.2 Normality Results Discussion

During PS-I, the author performed the test for normality using SPSS to verify if normality test conditions are valid or not.

1. Normality condition-1 Means and medians are not similar: This condition is not valid. Table 25 shows that the mean and median are similar.
2. Normality condition-2 Means are higher than standard deviations: This condition is valid as shown in Table 25.
3. Normality condition-3 Normality tests show all p-values are higher than 0.05: This condition is valid as shown in Table 26.
4. Normality condition-4 Histogram must show perfect bell curve: This condition is not valid as shown in Figure 39; the histogram does not show a perfect bell-curve.
5. Normality condition-5 Data points will be close to the diagonal line in Q-Q Plot. This condition is not valid as shown in Figure 40; the data points are not closed to the diagonal line.

From these above conditions, it can be concluded that the data is not normally distributed as not all five conditions are correct.

6.3.4.3 Mann-Whitney Test

Mann-Whitney test is one of the most common tests used to analyse non-parametric and small sample size data. It was performed to measure the significant differences between computing and ethical hacking students' scores in order to identify the students understanding of recurring security flaws, and whether they know how these flaws can be exploited.

The results obtained from the Mann-Whitney test analysis of software developers and ethical hackers scores can be compared in Table 27. It can be seen from the data in ranks table that the software developers score about 150 points in Q4, about 125 points in Q5 and about 163.5 points in Q7 more than ethical hackers, due to the difference in sum of ranks. This is a significant difference in mean scores between ethical hackers and software developers as p=.005, .009 and .035 respectively.

A Mann-Whitney test showed that there was a statically significant difference in Q4, Q5 and Q7 scores between software developers and ethical hackers as shown in Table 28, it is interesting to note that software developers

perform well in those questions that related to coding in contrast to ethical hackers.

Based on the results, it stated that software developers (computing related degree students) are more aware than ethical hackers of those vulnerabilities that occur during writing code because all these questions are related to implementation or coding phase.

Ranks

	Develop_EH	N	Mean Rank	Sum of Ranks
Q1	1	22	14.64	322.00
	2	6	14.00	84.00
	Total	28		
Q2	1	22	14.43	317.50
	2	6	14.75	88.50
	Total	28		
Q3	1	22	13.25	291.50
	2	6	19.08	114.50
	Total	28		
Q4	1	22	12.64	278.00
	2	6	21.33	128.00
	Total	28		
Q5	1	21	11.98	251.50
	2	6	21.08	126.50
	Total	27		
Q6	1	22	13.89	305.50
	2	6	16.75	100.50
	Total	28		
Q7	1	22	12.93	284.50
	2	6	20.25	121.50
	Total	28		
Q8	1	22	13.02	286.50
	2	6	19.92	119.50
	Total	28		
Q9	1	22	13.34	293.50
	2	6	18.75	112.50
	Total	28		
Q10	1	22	13.23	291.00
	2	6	19.17	115.00
	Total	28		

Table 27 Ranks table

Mann-Whitney Test Statistics[a]

	Q1	Q2	Q3	Q4	Q5	Q6
Mann-Whitney U	63.000	64.500	38.500	25.000	20.500	52.500
Wilcoxon W	84.000	317.500	291.500	278.000	251.500	305.500
Z	-.182	-.093	-1.764	-2.788	-2.623	-1.054
Asymp. Sig. (2-tailed)	.855	.926	.078	**.005**	**.009**	.292
Exact Sig. [2*(1-tailed Sig.)]	.892[b]	.935[b]	.126[b]	.020[b]	.010[b]	.460[b]

Mann-Whitney Test Statistics[a]

	Q7	Q8	Q9	Q10
Mann-Whitney U	31.500	33.500	40.500	38.000
Wilcoxon W	284.500	286.500	293.500	291.000
Z	-2.111	-1.950	-1.543	-1.623
Asymp. Sig. (2-tailed)	**.035**	.051	.123	.105
Exact Sig. [2*(1-tailed Sig.)]	.052[b]	.068[b]	.157[b]	.126[b]

Table 28 Mann-Whitney test statistics

a. Grouping Variable: Develop_EH

b. Not corrected for ties.

6.3.4.4 Results Discussion of Mann-Whitney Test

Results of the Mann-Whitney test indicate that computing students were aware of code level vulnerabilities such as

- Q4: Use of Externally-Controlled Format String Vulnerability
- Q5: Buffer Overflow Vulnerability
- Q7: Use of Dangerous Function Call in PHP Vulnerability

For the following vulnerability questions, the two-tailed p-value: was .005, .009 and .035. $p<0.05$; therefore, these are significant results as shown in Table 28.

From this data, it can be concluded that ethical hackers scores were significantly higher than software developers, except in Q4, Q5, and Q7 where ($U = 25$, $p = .005$, $U=20.5$, $p=.009$, and $U=31.5$, $p=.035$). Therefore, these are significant results.

6.3.5 Assessment of Questionnaire Internal Consistency

6.3.5.1 Cronbach's Alpha

Cronbach's Alpha is a measure used to determine the scale reliability in the questionnaire. For example, the pilot-study-I had included 10 questions to measure computing and ethical hacking students' ability to identify the vulnerabilities during the software development process. Let us consider the four scale items that used to measure participants' vulnerability understanding ranging from "not aware" to "well aware". A Cronbach's alpha was run on a sample size of 30 students.

Reliability Statistics

Cronbach's Alpha	Cronbach's Alpha Based on Standardized Items	N of Items
.844	.845	10

Table 29 Reliability statistics

6.3.5.2 Results Discussion of Cronbach's Alpha

As shown in Table 29, the alpha coefficient for the ten items is .844, suggesting that the items have relatively high internal consistency. (Note that a reliability coefficient of .70 or higher is considered "acceptable" in most social science research situations.).

A reliability analysis was carried out on the perceived task values scale comprising 10 Items. Cronbach's alpha showed the questionnaire to reach acceptable reliability, α =0.84. Most items appeared to be worthy of retention, resulting in a decrease in the alpha if deleted. See appendix Section 2.1 for correction and other related results.

6.3.6 Pilot-Study-I Overall Results Summary

There is a multitude of factors in the empirical literature, and because it is impossible to control all factors, it is often very difficult to get a meaningful result. The results of the PS-I, although statistically significant, due to a very sample size it is difficult to conclude that participants struggle to identify the vulnerabilities or lack sufficient understanding of recurrent vulnerabilities.

The primary goal of PS-I was to investigate the participants (software developers) ability to identify the software development errors that lead to vulnerabilities. Participants were given vulnerable code samples or UML class diagrams and asked to identify development errors. Exploring the participants' ability to know how malicious hackers could exploit these vulnerabilities was difficult due to the fact that all participants studied the same courses relating to software development and programming languages in the 2nd year. Thus, initially, all of them were considered to be software developers. However, after analysing participants' results, as shown in Figure 38 that students studying a cybersecurity major performed better than students studying with a computing major. The problem here is that students with a computing major learnt more about programming languages and software development modules in comparison to students with a cybersecurity major who learnt more about ethical hacking and computer security modules. Therefore, the PS-I split the participants into two groups: penetration testers (ethical hackers) and software developers (computing).

The statistically significant result has shown that ethical hacking students' scores were significantly higher than computing students' scores. This will be discussed in detail later in the discussion chapter. The results of this case study, therefore, raise the concern that computing students lack awareness of vulnerabilities and do not know how malicious hackers can exploit these. This case study also highlighted another fact that ethical hacking students lack the understating of coding level vulnerabilities.

It can be concluded that software developers are different from ethical hackers due to their background knowledge and working perspectives to achieve such as software developers worked to develop software, however, ethical hackers worked as penetration testers in order to find security flaws or

vulnerabilities. Both groups: penetration testers and software developers do not share common ground knowledge in order to develop or exploit software.

The results from PS-I raised some concerns and justified further investigation in PS-II such as vulnerable code samples would need to understandable and easy to explain the target vulnerability, as discussed in the following chapter PS-II. The study also had a number of limitations, namely:

- Not all computing relating degree students were software developers
- Sample size was very small.
- Experiment was very lengthy and time consuming with difficulty to understand vulnerable code samples.
- Participants were not aware of all vulnerabilities because of lack of awareness of some of the programming languages used in the questionnaire.

Based on these limitations, the next study was designed to avoid some of these. In particular, we used a language that all participants would be familiar with. The number of vulnerabilities to be examined was halved. The next study is described in Chapter 7.

7 Pilot Study–II (PS-II)

7.1 Introduction

7.1.1 General Description

The second pilot study aimed to analyse the effectiveness of the proposed Vulnerability Anti-Pattern (VAP), for software developers, both in terms of providing an effective understanding of vulnerabilities and an awareness of how to mitigate them. Furthermore, the study analysed the experimental results to determine the statistical significance to confirm the proposed hypothesis and ascertain its usefulness as a representative model. Through the process of vulnerabilities' awareness, recommendations would be that software developers could be better informed about recurrent vulnerabilities and their mitigation, without having to be an expert in cybersecurity.

7.1.2 Key Objectives

The Study Questions (SQ) are described in the following table

SH-1	**Do software developers have an effective understanding of recurrent vulnerabilities?**
SH-2	Can interventions based on Vulnerability Anti-Pattern help developers in improving their understanding of vulnerabilities?

7.1.3 Experiment Hypotheses

The study posits the following hypotheses (Experiment hypothesis=EH):

Experimental Study	
EH-1	The intervention study, which is based on Vulnerability Anti-Pattern, will improve participants' ability to identify the root causes of vulnerabilities during the different stages of the software development process
EH-2	The intervention will improve participants' ability to recognise and classify vulnerabilities using the terminology of the security community.
EH-0	There is no significant difference in participants' ability to identify the root-causes of vulnerabilities with and without intervention.
EH-3	There is no difference between "formal" and "informal" intervention in order to raise awareness of vulnerabilities.

EH-0	There is no significant difference between “formal” and “informal” intervention in order to raise awareness of vulnerabilities.
EH-4	Software developers will be able to retain awareness of vulnerabilities through interventions after a gap of one week.
EH-0	There is no significant difference in software developers’ obtained scores in the stage3 after a gap of one week.
Control Study	
EH-5	There is a significant difference between the performance of participants provided with intervention and those not provided with intervention.
EH-0	There is no significant difference between the performance of participants provided with intervention and those not provided with intervention.

7.2 Method

The experimental study and control study both followed the same method. The experimental study used intervention; however, the control was carried out without intervention. The sample size of both studies is not the same, which discussed further in limitation section 9.2.2.

7.2.1 Study Description

Experimental study designs, also called interventional study designs, are those where the researcher intervenes at some point throughout the study. A pre-post study survey measures the occurrence of an outcome before and after a particular intervention is applied. Both studies consisted of three sets of questionnaires, each of which consisted of pre-set and open-ended questions designed to investigate the core understanding and mitigation knowledge regarding the most commonly occurring software errors (Vulnerabilities). All participants were studying computing-related degrees such as Computing, Computer Games Technology and Computer Games Application Development.

The sample dataset was from a series of survey exercises conducted with 3rd and 4th year students of **the School of Design and Informatics at the**

University of Abertay Dundee. Overall, 39 participants took part in three stages of survey exercises.

1) Pre-Assessment Survey Study
2) Post-Assessment Survey Study
3) Post-Post-Assessment Survey Study

7.2.2 Studies Design Structure

The rationale for the experimental design is displayed in Table 30, which is comprised of four stages:

1. **Stage1: Pre-Assessment Survey**

Accessing and measuring participants (software developers) actual knowledge about prevalent vulnerabilities

2. **Intervention-Stage: Security Training Session**

Providing security training through the Vulnerability Anti-Pattern (Formal and Informal)

3. **Stage2: Post-Assessment Survey**

Accessing and measuring improvement or decline in participants (software developers) actual knowledge about prevalent vulnerabilities after the secure training

4. **Stage3: Post-Post-Assessment Survey**

Evaluating participants' long-term memory to show how much information they retained from the stage-2 provided secure training, after a gap of one week.

		Group A	Group B
Stage-1	Pre-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate participants results to measure their awareness of vulnerabilities	
Intervention-stage	Intervention Session		
	Input	Group A-Formal Vulnerability Anti-Pattern	Group B-Informal Vulnerability Anti-Pattern
	Output	Provide information about vulnerabilities	
Stage-2	Post-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate participants results to measure improvement/ deterioration after intervention provided by VAPs	
After 1 Week (one week gap)			
Stage-3	Post-Post-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate results to measure how much participants were able to retain the information from the provided intervention based on VAPs after a gap of one week.	

Table 30 Description of the Vulnerability Anti-Pattern experimental study structure, including a description of the inputs and outputs of all stages inputs.

7.2.3 Survey Questions' Structure

Table 31 details the question used during the experiment, which was essentially constructed based on accumulated necessary information from both communities' sources - software engineering and cybersecurity. The design of each question primarily demonstrated the connection of vulnerabilities root-causes with the Software Development Lifecycle (SDLC) phases from where they originated rather than categorised all of them as a coding error. It could be better

to find out why this error has occurred due to one of the following reasons: requirement specification phase error, design phase error or implementation phase error. Subsequently, this directs developers' attention towards the SDLC phase from which this vulnerability initiated. Each question was designed to ask software developers about the following necessary information. Part-1 of each question assessed participants' actual knowledge of the vulnerability while providing vulnerable codes or UML diagrams, Part-2 investigated participants' understanding of misuses or exploitations, and Part-3 inspected participants' ability to recognise and classify the vulnerabilities using the terminology of the cybersecurity community. The question structure is as follows:

1. Part-1: Vulnerable code or UML diagram
2. Part-2: Misused or exploited explanation
3. Part-3: Identify vulnerability formal name

Each questionnaire consisted of five sets of questions, and each set of questions investigated the developers' knowledge to find a vulnerability. The selected vulnerabilities were chosen from the "2011 CWE/SANS Top 25 Most Dangerous Software Errors" list. Of 90 students doing to computing degrees, 39 successfully completed the questionnaire. Each question was divided into three sub-parts:1) identifies developer (students) awareness of the provided vulnerable code or UML diagram; 2) investigated vulnerability root-cause description in a formal and informal way; 3) explored related attack pattern and exploitation knowledge.

Part-1:
Vulnerable code or UML diagram
	Use the PHP code sample below to answer questions I, II and III below.
	1. $id = $_COOKIE["mid"] ;//Assign cookie value to id variable
	2. mysqli_query("SELECT MessageID, Subject FROM messages WHERE MessageID = '$id'");// mysqli query to retrieve message number(id)
	The program output will print a message that corresponds to the message ID.
	The above code contains a flaw, which may not be detected by the compiler. Please describe below why you think this code is wrong.

Part-2: Misused or exploited explanation	Use the PHP code sample below to answer questions I, II and III below.
	1. $id = $_COOKIE["mid”] ;//Assign cookie value to id variable
	2. mysqli_query("SELECT MessageID, Subject FROM messages WHERE MessageID = '$id'");// mysqli query to retrieve message number(id).
	The program output will print a message that corresponds to the message ID.
	A malicious hacker could exploit this code because it lacks input validation on $id and can be used to gain access to the database information. Can you explain how this code would enable a malicious hacker to carry out such an attack?

Part-3: Identify vulnerability formal name	Use the PHP code sample below to answer questions I, II and III below.
	1. $id = $_COOKIE["mid”] ;//Assign cookie value to id variable
	2. mysqli_query("SELECT MessageID, Subject FROM messages WHERE MessageID = '$id'");// mysqli query to retrieve message number(id).
	The program output will print a message that corresponds to the message ID.
	In your opinion, which of the following best describes the flaw (tick all that apply)?
	□ Injection Flaw
	□ Improper Input Validation
	□ SQL injection
	□ Information leakage
	□ String format error

Table 31 PHP sample question

Another example of a question used during the experiment.

Use the C# code sample below to answer the following questions I, II and III?

```
1. string userName = ctx.getAuthenticatedUserName(); //function call to
   get user name
2. string query = "SELECT * FROM items WHERE owner = '"
   + userName + "' AND itemname = '" + ItemName.Text + "'"; // database
```

query to retrieve item information using user name and itemname as an input value from user. 3. sda = new SqlDataAdapter(query, conn);// execute the query The program output will retrieve item information that corresponds to the username and itemname.
I. The above code contains a flaw, which may not be detected by the compiler. Please describe below why you think this code is wrong.
II. A malicious hacker could exploit this code to concatenate malicious input to build SQL command to skip any input validation on username and can be used to gain access to database information. Can you explain how this code would enable a malicious hacker to carry out such an attack?
III. In your option, which of the following best describes the flaw (tick all that apply)? Injection Flaw
□ Improper Input Validation
□ SQL injection
□ Information leakage
□ String format error

Table 32 Experiment question related to C#

Tables 31 and 32 represent the questions used during the experiment. The answers of each participant have been marked as follows:

- 0 = Student is unaware of the vulnerability.
- 1 = student is slightly aware of the vulnerability
- 2 = student is aware of the vulnerability
- 3 = student is well aware of the vulnerability

7.2.4 Questionnaire Design

PS-II included questions based on the vulnerable code, which were written by the researcher with the help of ethical hacking, PHP and C\C++ lecturers. The research considered the students' (participants') level of understanding and degree of code complexity during the questionnaire design. For generating

vulnerable codes, a penetration training environment was used such as OWASP Mutillidae II (Druin 2011) and bWAPP (Happel 2017).

7.2.5 Security Intervention Session

As a part of the knowledge awareness process (KAP), explained in Section 3.4.1.3, an intervention[8] was provided, based on the VAP, during the intervention-stage. The intervention stage was designed to measure VAPs' effectiveness in subsequent stages: post-assessment and post-post-assessment as shown in Figure 41. There are two types of interventions: formal and informal, which were evaluated in

- Post-Assessment stage, just after it was provided.
- Post-Post-Assessment stage, after a gap of one week.

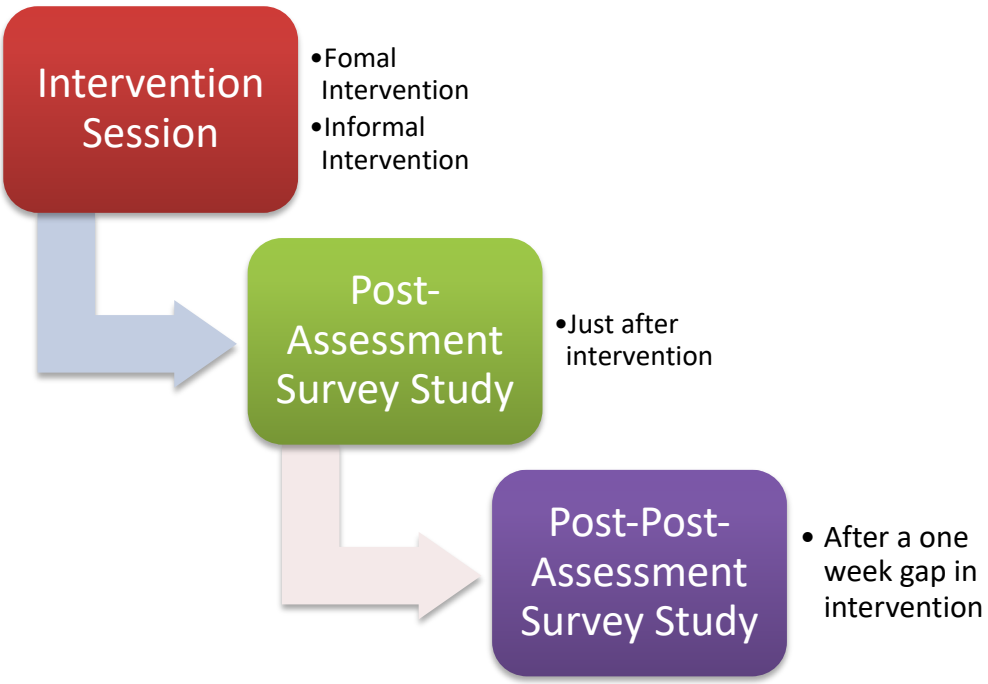


Figure 41 Security intervention link with assessment phases

[8] Researcher had provided the intervention to intervene at stage-2 throughout the study. A pre-post study measures the occurrence of an outcome before and again after a VAPs based intervention is implemented.

7.2.6 Security Intervention Types

The intervention, which is based on the concept of Vulnerability Anti-Patterns, was provided to participants during the experiment. As shown in Figure 42, the intervention was designed in two formats:

- **Formal Vulnerability Anti-Pattern:** the reason to call it formal is that the standard "Anti-Pattern" format has been followed while designing the formal version. Through the formal Vulnerability Anti-Pattern, software developers may be able to raise awareness of vulnerabilities that include problem descriptions relating to vulnerability databases (CWE, CVE), vulnerability risk patterns pointing to software fault patterns, consequences and countermeasures offered by security patterns included in code examples.
- **Informal Vulnerability Anti-Pattern:** it followed an informal way of presenting similar information to the formal version. A vulnerable scenario was presented in the informal version because software developers would gain awareness of the vulnerability through a detailed example, which demonstrated the root-causes, consequences and countermeasures of the vulnerability.

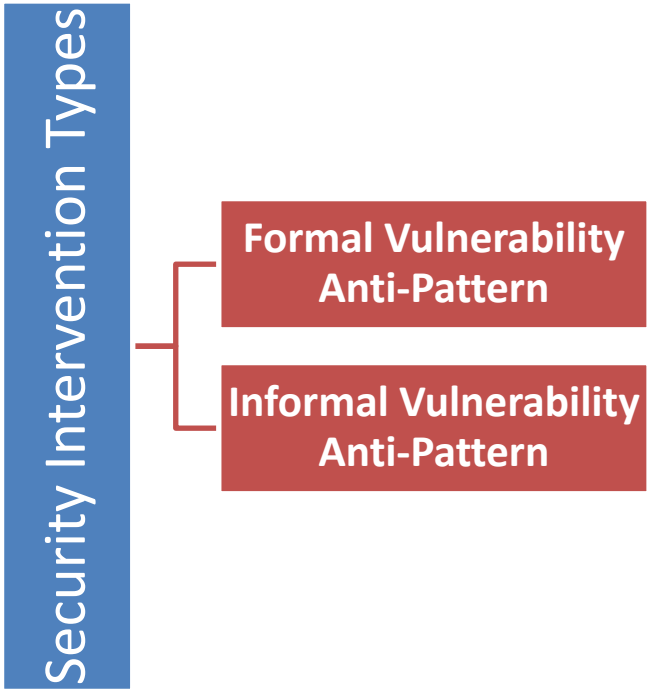


Figure 42 Security intervention types

Using two types of security intervention, the study evaluates the following hypothesis:

EH-3: *There is no difference between "formal" and "informal" intervention in order to improve awareness about the particular vulnerability.*

7.2.7 Control Experiment Description

The goal of the control experiment was to demonstrate the effectiveness of the intervention. The control experiment was carried out with 14 participants. The experimental structure was identical but excluded the intervention stage. Control participants did not receive any intervention. The study measured the scores' difference between the experimental and control groups as shown in Table 33, which detailed in Section 7.4.

Experimental Group	Control Group
Stage-1 without intervention	Stage-1 without intervention
Stage-2 with intervention	Stage-2 without intervention
Stage-3 with intervention	Stage-3 without intervention

Table 33 Experimental and control group comparison

7.2.8 Vulnerability Sample Size

The selection of vulnerabilities was purely based on those programming languages, which students already taught and were well aware of. The module lecturers were consulted to ensure that experiment questions were understandable and not too complicated for the students. During PS-I, 10 vulnerabilities were tested with students. This study's feedback mentioned that the experiment was too lengthy and time-consuming. Consequently, in the PS-II only 5 vulnerabilities were selected.

This study considered the NVD to be the primary source of data relating to vulnerabilities, which included CVE as a sub-set repository to trace and tracks the most serious vulnerabilities. This research aims to raise awareness of those software errors that had serious and dangerous consequences for the system. Five vulnerabilities were selected from the list of '2011 CWE/SANS Top 25 Most dangerous software errors', which is purely based on students' knowledge (software developers) of the programming language.

The students had different background knowledge, depending on their degree choice. They are divided into two categories: Computing and Gaming, based on their expertise and competence in programming languages.

- **Computing students:** The experiment included the five vulnerabilities for the 'COMPUTING' students in Table 34.

Question Number	Vulnerability OWASP Rank	Vulnerability Name	Stage-1	Stage-2	Stage-3	Example Code Language
Q1	Rank 1	SQL Injection	√	√	√	PHP, C#
Q2	Rank 5	Missing Authentication	√	√	√	UML Class Diagram
Q3	Rank 6	Missing Authorization	√	√	√	PHP
Q4		Deprecated Function Call	√	√	√	PHP
Q5	Rank 24	Integer Overflow	√	√	√	PHP

Table 34 Survey summary for the computing students, its included vulnerabilities and vulnerable code or UML diagram description

- **Gaming students:** The experiment included the following five vulnerabilities for the 'GAMING' students in Table 35.

Question Number	Vulnerability OWASP Rank	Vulnerability Name	Stage-1	Stage-2	Stage-3	Example Code Language
Q1	Rank 5	Missing Authorization	√	√	√	UML Sequence Diagram
Q2	Rank 3	Buffer Overflow	√	√	√	C++

Q3	Rank 18	Use of Dangerous Function Call	√	√	√	C++
Q4	Rank 24	Integer Overflow	√	√	√	C++
Q5	Rank 20	Incorrect Calculation of Buffer Size	√	√	√	C++

Table 35 Survey summary for the gaming students, its included vulnerabilities and vulnerable code or UML diagram description

- **Vulnerability Anti-patterns:** As shown in Table 36, the experiment included the proposed design "Vulnerability Anti-Patterns" as a security intervention for gaming and computing students.

#	Vulnerability Anti-Pattern	Computing	Gaming	Example Code Language
1	SQL Injection Vulnerability Anti-Pattern	√		PHP
2	Missing Authentication for Critical Functions Vulnerability Anti-Pattern	√		PHP
3	Missing Authorization Vulnerability Anti-Pattern	√	√	PHP
4	Buffer Overflow Vulnerability Anti-Pattern		√	C/C++
5	Use of Deprecated Function Vulnerability Anti-Pattern	√		PHP
6	Use of Potentially Dangerous Function Vulnerability Anti-Pattern		√	C/C++
7	Integer Overflow Vulnerability Anti-Pattern	√	√	PHP, C/C++

8	Incorrect Calculation of Buffer Size Vulnerability Anti-Pattern		√	C/C++

Table 36 Used Vulnerability Anti-Patterns as security intervention for computing and gaming students

7.2.9 Participant Sample Size

In this experiment, the term "software developers" will be used in its broadest sense to refer to all students of Computing, Computer Games Technology and Computer Games Application Development. Table 37 presents participants' information about their Numbers, Degree Title, Year of Study and stages in which they had participated.

Number of Participants	Degree Title	Stage-1	Stage-2	Stage-3
8	Computer Games Application Development		√	√
5	Computer Games Application Development	√	√	√
4	Computer Games Technology	√	√	√
12	Computer Games Technology		√	√
5	Computing	√	√	√
5	Computing	√	√	√

Table 37 Participants information

7.3 Experimental Study

7.3.1 Research Hypotheses

The statistical analysis evaluates two successive measurements of paired samples to know whether their means are significantly different (Table 38).

Research Question	**Does intervention, based on the use of "formal or informal Vulnerability Anti-Pattern", improve participants' ability to identify the root-causes of vulnerabilities?**	
EH-1	The intervention will improve participants' ability to identify the root causes of vulnerabilities during different stages of the development process.	
EH-2	The intervention will improve participants' ability to recognise and classify vulnerabilities using the terminology of the security community.	
EH-0	There is no significant difference in participants' ability to identify the root-causes of vulnerabilities with and without intervention.	
Depended Variables	Total_Score_stage-1	Total_Score-stage-2
Independent Variables	DegreeCode 1=Computing 2=Gaming	

Table 38 Research questions and hypotheses: two samples test

7.3.2 Examining Significant Difference between Two Scores Samples

The dataset consisted of participants' scores (software developers), called Total_Score_stage-1 and Total_Score_stage-2. Each participant completed the predefined questionnaire twice during the experiment study: Firstly, without any security training; secondly, after the secure training. They were marked based on their answers and a total score of each stage was calculated. Thus, Total_Score_stage-1 was a total obtained score by a participant in stage-1 and Total_Score_stage-2 was a total obtained score by a participant in stage-2.

Specifically, we are interested in the *difference* between Total_Score_stage-1 and Total_Score_stage-2 for each participant. In other words, the intention was to determine whether participants performed better after receiving the intervention in order to recognise and identify vulnerabilities in stage-2 than their first time in stage-1.

7.3.2.1 Key Terms

- **Intervention:** this process raises security awareness. It comprised of the Vulnerability Anti-Pattern (a proposed solution to educate the software developers of recurrent vulnerabilities).
- **Independent variable:** degree_code, 1=Computing, 2= Gaming
- **Dependent variables:** Total_score_stage1, Total_score_stage2

7.3.3 Descriptive Statistical Analysis

Before performing any formal statistical tests, it was necessary to perform a detailed exploration and description of the sample data through descriptive analysis. For the sample data, this was an essential preliminary step. For that reason, during the descriptive statistical analysis, the sample data was explored thoroughly, while including a wide range of useful statistics, i.e. exploration of frequencies, mean values and median values. To supplement the descriptive statistics, a graphical representation (Histogram) was included to provide a validation of the data. The study included the following descriptive statistics:

1. Mean, median and mode
2. Frequency differences between the two samples
3. Histograms

1. Compare the Mean, Median and Mode

As shown in Table 39, the descriptive analysis of the total_stage-1 and total_stage2 compared the mean value, median value and mode value. A total of 18 participants, data were collected, one missing value that was not excluded. Total_stage1 score (score obtained by participants in the stage1) mean value was 6.72 and median value 7.00, in contrast to total_stage2 score (score obtained by participants in the stage2) mean value of 11.72 and the median value of 11.500.

Thus, it is apparent from Table 39 that participants scored double with average mean value 11.7222 in stage 2, which was performed after the intervention study.

Mean, Median and Mode Statistics

		TOTAL_stage1	TOTAL_stage2
N	Valid	18	18
	Missing	1	1
Mean		**6.7222**	**11.7222**
Median		**7.0000**	**11.5000**
Mode		7.00	11.00
Std. Deviation		1.84089	1.36363

Table 39 Compared the mean, median and mode of Total_stage1 and Total_stage2 and presented the total number of participants

2. Frequency Table

The difference in frequencies was analysed and the central tendency across the stage-1 and the stage-2 scores (Total_stage1 and Total_stage2) calculated. Table 40 displayed the numbers of participants in stage-1. Those obtained scores ranging from 4 to 10. Table 40 shows that frequently obtained score by participants during the stage-1 was 7 (also known, as the central tendency of Total_stage1 was 7). The minimum score value was 4 and the maximum score was 10.

TOTAL_stage1

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	4.00	2	10.5	11.1	11.1
	5.00	4	21.1	22.2	33.3
	6.00	1	5.3	5.6	38.9
	7.00	**6**	**31.6**	**33.3**	**72.2**
	8.00	2	10.5	11.1	83.3
	9.00	1	5.3	5.6	88.9
	10.00	2	10.5	11.1	100.0
	Total	18	94.7	100.0	
Missing	System	1	5.3		
Total		19	100.0		

Table 40 Numbers of participants in the stage1.

Table 40 presents participants' frequencies with respect to their obtained scores in stage-2. The highest obtained score was 15 and the lowest 9. Table 41 shows that a median of obtained scores during the stage-2 was 11 (also known, as the central tendency of Total_stage-2 was 11).

TOTAL_stage2

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	9.00	1	5.3	5.6	5.6
	10.00	1	5.3	5.6	11.1
	11.00	7	36.8	38.9	50.0
	12.00	4	21.1	22.2	72.2
	13.00	4	21.1	22.2	94.4
	15.00	1	5.3	5.6	100.0
	Total	18	94.7	100.0	
Missing	System	1	5.3		
Total		19	100.0		

Table 41 Numbers of participants in the stage2.

Data from Table 40 can be compared with data in Table 41, which clearly demonstrates that the stage-1 maximum score is equal to the stage-2 minimum obtained score. These results suggest that the security training method had considerably improved participants' ability to identify the root causes of vulnerabilities during different stages of the development process. The themes identified in these responses can be compared by both stages' average means, which illustrates a substantial difference in their mean values such as Stage-1=6.7222 and Stage-2=11.7222 in Table 39.

3. Histograms

Figures 43 and 44 show stage-1 and stage-2 score distributions. On the x-axis, the total score of each stage is displayed and on the y-axis, the frequency (numbers of participants) score obtained was presented. The graphical presentation of two histograms shows that both stages' maximum frequency was almost the same, although there is a tendency for participants to score higher in stage-2 after receiving the intervention, in comparison to stage-1. In addition, after comparing the total score of both stages, it can be noticed in their mean and standard deviation values that participants appear to have been performed differently before and after receiving the intervention. Overall, these results indicated that the intervention had considerably improved participants' ability to identify the root causes of vulnerabilities.

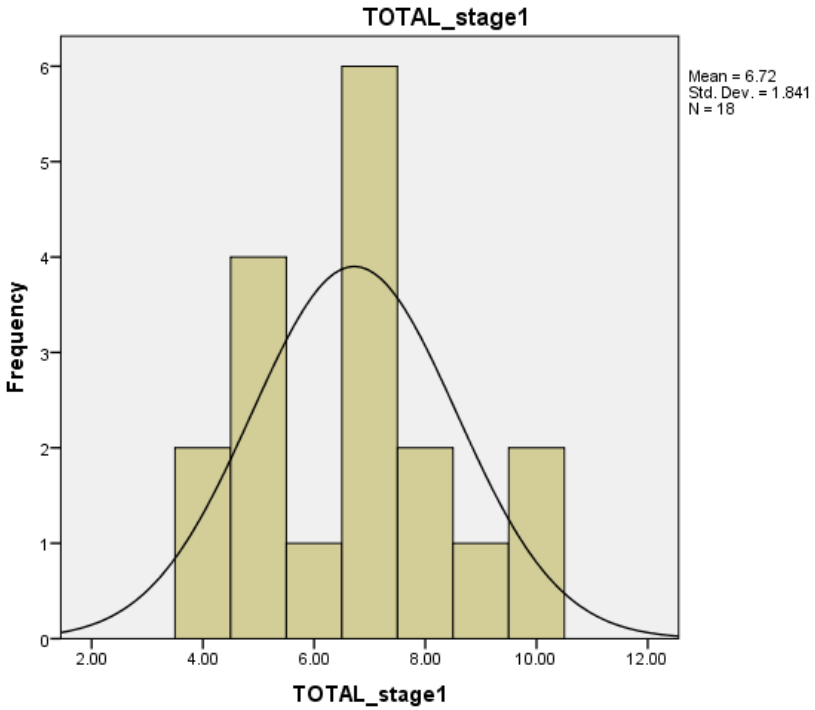


Figure 43 Showing distribution of scores in stage-1 participants

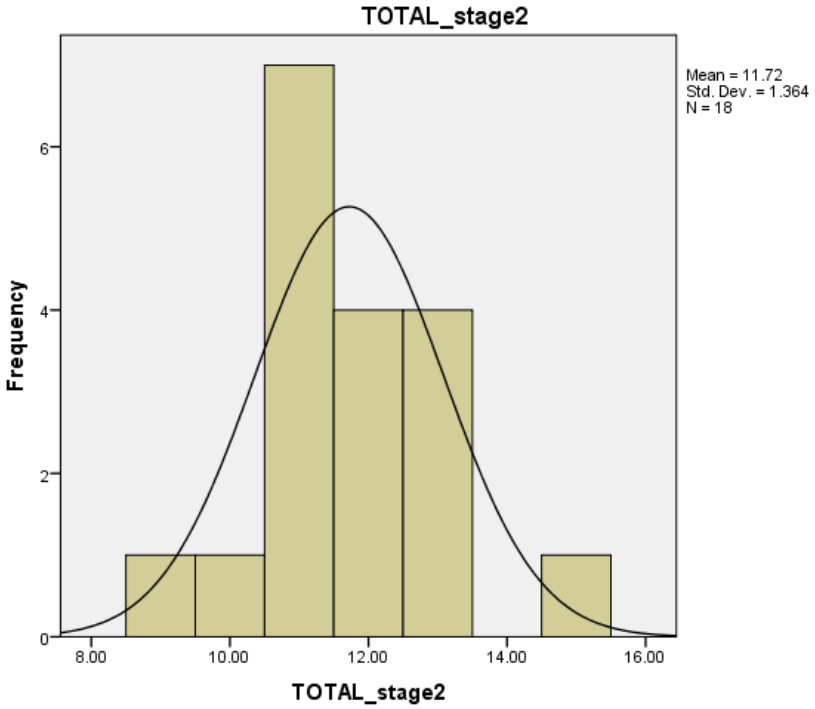


Figure 44 Showing the distribution of scores in stage-2 participant

7.3.3.1 Selection of Appropriate Statistical Test for Related Sample Data

To explore the significance of the difference between the two sample sets of scores, the study performed a statistical test. As a guide to choosing an appropriate statistical test for the sample data, Figure 45 presents a flow chart, which can be used to show researchers how to implement the recommended tests when there are two samples of scores. This is done in order to compare students' scores to measure the effectiveness of the 'VAPs'.

The observation has paired for each subject (participants) in the sample and two successive measurements had performed with the same set of participants without and with intervention (security training). As shown in Figure 45, this study had chosen a Paired-Samples t Test. The detail of the decision process, which included the explanation of why the Paired-Samples t Test was chosen, has been described further in the below sub-sections.

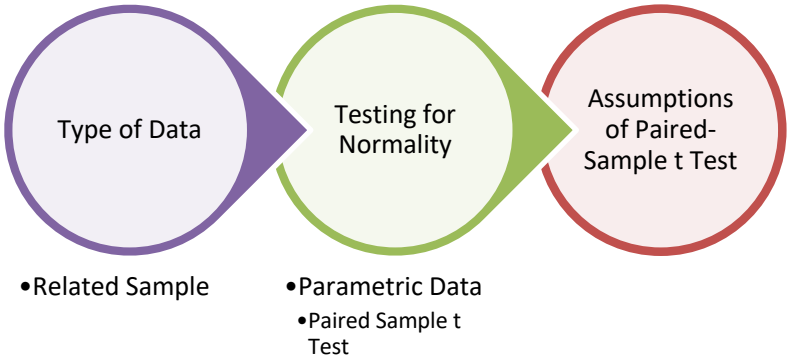


Figure 45 Choosing the appropriate statistical test for related two samples to measure the effectiveness of the proposed intervention study

7.3.3.2 Type of Data: Related Samples

During statistical analysis, it is essential to know what the type of data we have, as we are aware that the same group of participants (software developers) performed two sets of questionnaires twice and, from them, two sets of total scores were generated, which clearly shows we need to categorise the sample data as a related samples or paired samples.

7.3.3.3 Testing for Normality

A normality test was used to determine if the data were normally distributed. There are two main methods of assessing normality:

1. Graphically: Histograms
2. Numerically: Test of normality

1. Histograms

The visual examination of both stages total scores data in the histograms, in Figures 46 and 47 show that data is normally distributed.

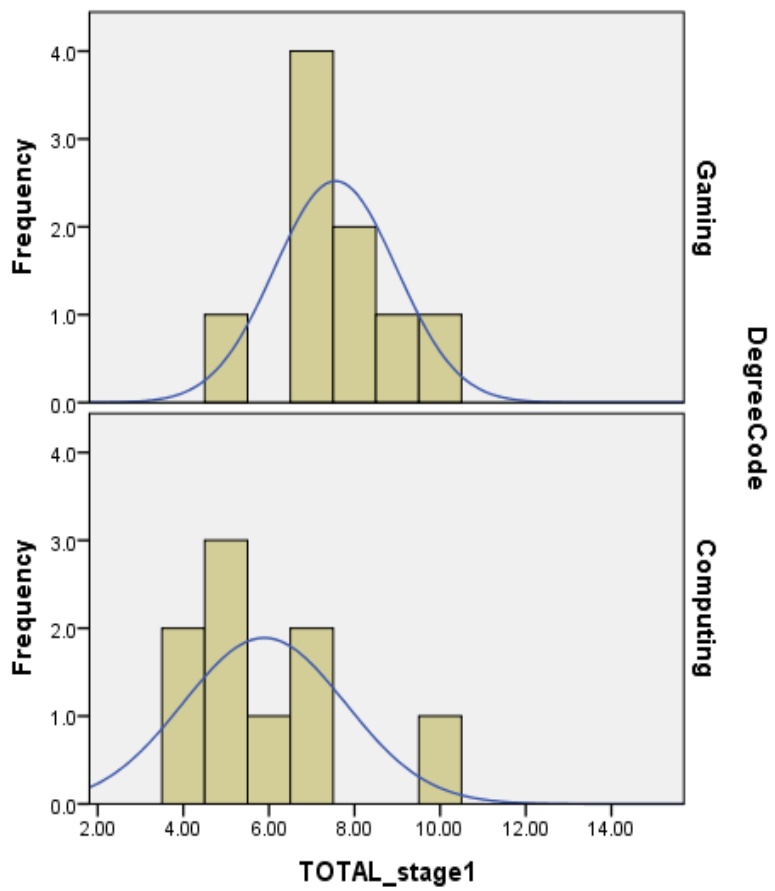


Figure 46 Total_Stage1 scores frequency regarding the students' degree

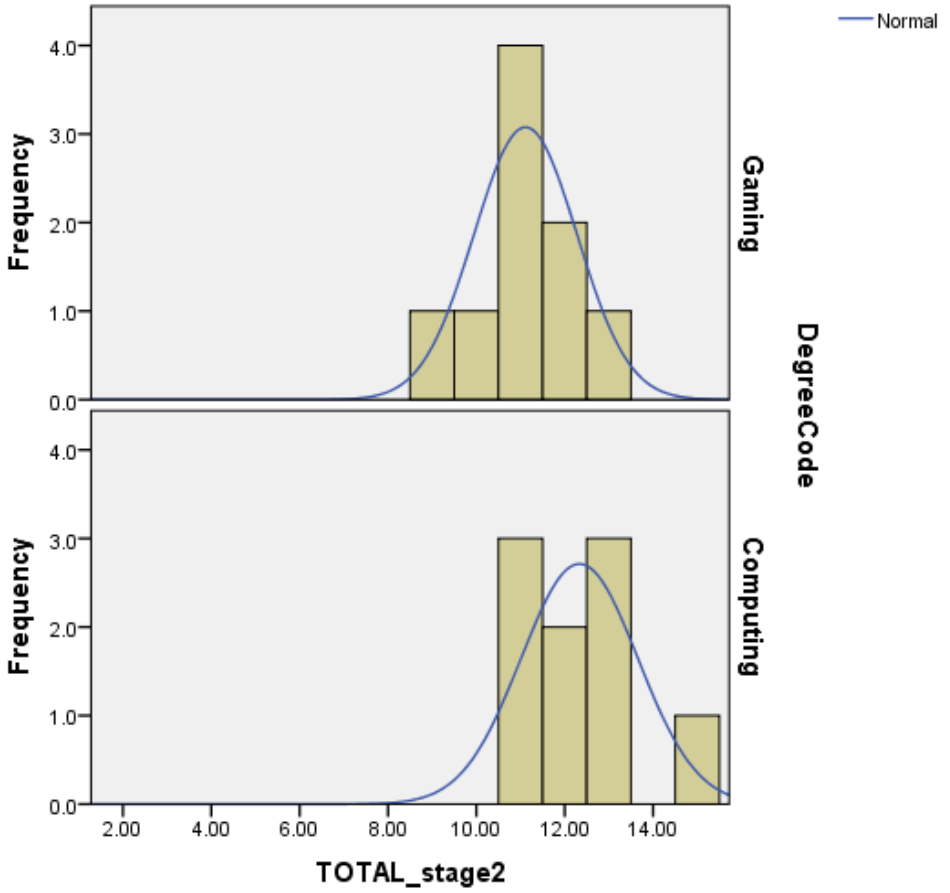


Figure 47 Total_Stage2 scores frequency regarding the students' degree

2. Test of Normality

Table 42 presents the results from two well-known tests of normality; namely the Kolmogorov-Smirnov Test and the Shapiro-Wilk Test. Because of the small sample size, which included only 18 participants with one missing value), the Shapiro-Wilk test was recommended as being more appropriate for this experiment (< 50 samples). For this reason, the author used the Shapiro-Wilk test numerical means for measuring normality.

From Table 42, it would be claimed that for Degree_code the dependent variable "Gaming" and "Computing", with the independent variables "Total_stage1" and "Total_stage2", data were normally distributed.

To validate the test of normality, it is vital to know that, if the Sig. value of the Shapiro-Wilk Test is higher than 0.05, then the data is normal. If it is below 0.05 then the data significantly deviates from a normal distribution.

Tests of Normality

		Kolmogorov-Smirnov[a]			**Shapiro-Wilk**		
	DegreeCode	Statistic	df	Sig.	Statistic	df	Sig.
TOTAL_stage1	Gaming	.237	9	**.154**	.930	9	**.486**
	Computing	.236	9	**.161**	.864	9	**.106**
TOTAL_stage2	Gaming	.240	9	**.144**	.941	9	**.595**
	Computing	.196	9	**.200***	.872	9	**.130**

Table 42 Normality tests shown in the 'sig columns'.

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

In Table 42 tests of normality, there were no significant results in the sig columns value that is equal or less than 0.05. Neither normality test includes any statistically significant results, and this has been confirming observations from the histograms in Figures 46 and 47. Taken together, these results suggest that the data was normally distributed.

7.3.3.4 Assumptions of Paired-Sample t Test

1. **Assumption #1:** Dependent variable should be measured on a continuous scale. The dependant variables are on a continuous scale such as Total_stage1 scores range from 0 to 3.
2. **Assumption #2:** Sample independent variables should consist of two categorical, "related groups" or "matched pairs". The sample independent variable consists of two categories “Gaming” and “Computing” participants
3. **Assumption #3:** There should be no significant outliers in the differences between the two related groups. There is no prominent outlier in the sample.
4. **Assumption #4:** The distribution of the differences in the dependent variable between the two related groups should be approximately normally distributed. The test of normality confirms this.

7.3.3.5 Paired-Samples t Test Discussion

The paired-samples t Test compares the means between two related groups on the same continuous, dependent variable. For example, the study uses the paired-samples t-test to measure whether there was a difference in participants (software developers) performance related to finding vulnerabilities during software development process before and after receiving the intervention (a secure training method). In this case, the dependent variable is “total score”, and related groups (gaming and computing) would be the obtained score values “before=Total_stage1” and “after=Total_tsage2” the intervention study.

The subject in the sample was the same and focused on the difference between two successive measurements: stage-1 (Pre-Assessment) score and stage-2 (Post-Assessment) score. Specifically, we were interested in the difference between the first and second scores for each participant. The test is performed to see if scores improved. To test the hypothesis the researcher had to perform the test. Therefore, in this test, the null hypothesis suggests that there was no improvement in the participants’ scores.

- **Stage-1-Pre-Assessment Survey Study:** Participants (software developers) were asked to answer a series of questions in order to measure their knowledge of common software errors (vulnerabilities). The stage-1 questions included SQL Injection (CWE-89), Buffer Overflow (CWE-120), Missing Authentication

for Critical Function (CWE-306), Missing Authorization (CWE-862), Use of Potentially Dangerous (Deprecated) function (CWE-676), Incorrect Calculation of Buffer Size (CWE-131).

- After receiving the “intervention”.
- **Stage-3-Post-Assessment Survey Study:** Participants (software developers) were asked to answer similar questions in order to measure how much they had improved in their knowledge about common software errors (vulnerabilities). The stage-2 questions followed the same format and included similar questions, such as SQL Injection (CWE-89), Buffer Overflow (CWE-120), Missing Authentication for Critical Function (CWE-306), Missing Authorization (CWE-862), Use of Potentially Dangerous (Deprecated) function (CWE-676), Incorrect Calculation of Buffer Size (CWE-131).

7.3.3.6 Result Discussion of Paired-Samples t Test

Table 43 provides summary statistics for the two conditions. This includes the sample size (N=18), mean, standard deviation and standard error mean for the dependent variables (Total_stage1 and TOTAL_stage2) for each condition of the independent variable (Gaming and Computing).

The detailed analysis included all questions to measure significant differences. The experiment hypotheses state that participants would score more after receiving the intervention, which was intended to help participants to identify the root causes of vulnerabilities, in contrast to participants’ scores before receiving the intervention. The null hypothesis is rejected, and participants improved their ability to identify the root causes of vulnerabilities after the intervention, during stage-2.

7.3.3.6.1 Paired Sample Statistics Table

Table 43 presents the mean number of participants and the standard deviation values of the variables. As shown in Table 43, there were six pairs: the first pair presented the difference between both stages’ total scores and the remaining pairs described the difference in both stages of each set of questions’ scores. The mean column defined all six pairs mean value differences, N column included the number of participants, and then the St. Deviation column explained the standard deviation differences.

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	TOTAL_stage1	6.7222	18	1.84089	.43390
	TOTAL_stage2	11.7222	18	1.36363	.32141
Pair 2	Q1-s1	1.22	18	1.060	.250
	Q1-S2	2.11	18	.963	.227
Pair 3	Q2-s1	1.22	18	.647	.152
	Q2-S2	2.06	18	1.056	.249
Pair 4	Q3-s1	1.75	16	1.291	.323
	Q3-S2	2.63	16	.500	.125
Pair 5	Q4-s1	1.39	18	.916	.216
	Q4-S2	2.56	18	.511	.121
Pair 6	Q5-s1	1.41	17	.712	.173
	Q5-S2	2.41	17	.618	.150

Table 43 Paired samples statistics for total and all questions scores

From Table 44, it would be suggested that there was a significant correlation between pair2 and pair6 (Q1 and Q5).

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	TOTAL_stage1 & TOTAL_stage2	18	.085	.739
Pair 2	**Q1-s1 & Q1-S2**	**18**	**.550**	**.018**
Pair 3	Q2-s1 & Q2-S2	18	.325	.188
Pair 4	Q3-s1 & Q3-S2	16	-.052	.849
Pair 5	Q4-s1 & Q4-S2	18	.014	.956
Pair 6	**Q5-s1 & Q5-S2**	**17**	**.584**	**.014**

Table 44 Paired samples statistics and correlations

Paired Samples Test

		Paired Differences			
		95% Confidence Interval of the Difference			
		Upper	t	df	Sig. (2-tailed)
Pair 1	TOTAL_stage1 - TOTAL_stage2	-3.90783	-9.659	17	.000
Pair 2	Q1-s1 - Q1-S2	-.410	-3.915	17	.001
Pair 3	Q2-s1 - Q2-S2	-.315	-3.389	17	.003
Pair 4	Q3-s1 - Q3-S2	-.125	-2.485	15	**.025**

Pair 5	Q4-s1 - Q4-S2	-.648	-4.745	17	.000
Pair 6	Q5-s1 - Q5-S2	-.685	-6.733	16	.000

Table 45 Result of the paired samples t test

Table 45 reports that the mean score during stage-2 (M= 11.7222, SD=1.36363) is significantly (t= -9.659 and p< 0.05) greater than the mean score of the same participants during stage-1 (M= 6.7222, SD=1.84089).

7.3.3.7 Conclusion

As t (17) = -3.9078, p < 0.001 and the mean values of the two score pairs values and from the direction of the t-value, it can be concluded that there is a statistically significant improvement in participants' scores after receiving the intervention, which ranges from 6.722 ± 1.84 to 11.72.52 ± 1.84 (p < 0.001). Hence, the null hypothesis is rejected, which indicates that there is no significant difference in participants' ability to identify the root-causes of vulnerabilities with intervention. Therefore, it can be concluded that the proposed intervention improved participants' ability to identify the root causes of vulnerabilities during different stages of the development process.

7.4 Research Hypothesis

Research Question	Is there a difference between "formal" and "informal" intervention in order to provide information of vulnerabilities?
EH-3	There is a difference between "formal" and "informal" interventions in order to provide information of vulnerabilities.
EH-0	There is no significant difference between "formal" and "informal" interventions in order to provide information of vulnerabilities.
Dependent Variables	Total_Score-Stage2
Independent Variables	Vulnerability Anti-Pattern Code 1=Formal 2=Informal
Key Terms	Formal training method= Formal Vulnerability Anti-Pattern Informal training method= Informal Vulnerability Anti-Pattern

Table 46 Research question and hypothesis

7.4.1.1 Descriptive Analysis of Total_Score_Stage2 Based on the Intervention Type

Table 47 presents a case processing summary, in which there were 39 participants. During stage2, participants trained using "Vulnerability Anti-Pattern" about prevalent vulnerabilities while using two types of formats: 1=Formal training method, 2=Informal training method. Out of 39, 18 participants had received the formal intervention and 21 participants had received the informal intervention.

Case Processing Summary

		Cases			
		Valid		Missing	
	VulnerabilityAntiPatternCode	N	Percent	N	Percent
Total_Score_Stage2	Formal	18	100.0%	0	0.0%
	Informal	21	100.0%	0	0.0%

Table 47 Case processing summary stage2

Simple statistical analysis was used to measure the mean of both types of interventions to evaluate the difference in participants' scores. The means of

total_Score_Stage2 for “Formal intervention” is 11.83, whereas the “Informal intervention” mean is 11.33. Thus, it can be concluded that there is no difference in participants mean scores as shown in Table 48.

Descriptives

	VulnerabilityAntiPatternCode			Statistic	Std. Error
Total_Score_Stage2	Formal	Mean		11.83	.390
		95% Confidence Interval for Mean	Lower Bound	11.01	
			Upper Bound	12.66	
		5% Trimmed Mean		11.93	
		Median		12.00	
		Variance		2.735	
		Std. Deviation		1.654	
		Minimum		8	
		Maximum		14	
		Range		6	
		Skewness		-.668	.536
		Kurtosis		.385	1.038
	Informal	Mean		11.33	.634
		95% Confidence Interval for Mean	Lower Bound	10.01	
			Upper Bound	12.66	
		5% Trimmed Mean		11.74	
		Median		12.00	
		Variance		8.433	
		Std. Deviation		2.904	
		Minimum		0	
		Maximum		15	
		Range		15	
		Skewness		-3.156	.501
		Kurtosis		12.403	.972

Table 48 Descriptive analysis

Table 49 presents the highest and lowest obtained scores during each type of intervention. Interestingly, the participant who got “informal intervention” case number 6 obtained full marks. Case number 10 is an outlier because this participant performed stage-1 but did not complete the questionnaire in stage-2.

Extreme Values

	VulnerabilityAntiPatternCode			Case Number	Value
Total_Score_Stage2	Formal	Highest	1	19	14
			2	20	14
			3	23	14
			4	2	13
			5	4	13[a]
		Lowest	1	21	8
			2	18	9
			3	22	11
			4	15	11
			5	13	11[b]
	Informal	Highest	1	6	15
			2	8	13
			3	29	13
			4	31	13
			5	33	13[a]
		Lowest	1	10	0
			2	25	9
			3	36	10
			4	26	10
			5	34	11[b]

Table 49 Highest and lowest scores’ table during stage-2

a. Only a partial list of cases with the value 13 are shown in the table of upper extremes.

b. Only a partial list of cases with the value 11 are shown in the table of lower extremes.

7.4.1.2 Selection of Appropriate Statistical Tests for the Sample Data

A normality test was performed similar to Section 7.3.4, Table 50 presents the results from two well-known tests of normality. In this case, the Shapiro-Wilk Test is more

appropriate for small sample sizes (< 50 samples). As can be seen from Table 51, the data normality assumption is correct.

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk
	VulnerabilityAntiPatternCode	Statistic	df	Sig.	Statistic
Total_Score_Stage2	Formal	.196	18	**.066**	.910
	Informal	.264	21	**.001**	.642

Table 50 Tests of normality

Tests of Normality

		Shapiro-Wilk[a]	
	VulnerabilityAntiPatternCode	df	Sig.
Total_Score_Stage2	Formal	18	.085
	Informal	21	.000

Table 51 Tests of normality

a. Lilliefors Significance Correction

7.4.1.3 One-Way ANOVA Test

Since the normality test indicated that stage-2 participants' data (after receiving the intervention) was normally distributed, a parametric type of test was required to analyse the statistical significance of results gained.

The one-way analysis of variance (ANOVA) is used to determine whether there are any statistically significant differences between the means of stage-2 participants; those received two different types of interventions. The test was run using a 0.05 alpha level, and was two-tailed, in a bid to detect an effect in either direction. To this effect, if a test achieves a $p < 0.05$, this was deemed to reflect a statistically significant difference between the two samples.

7.4.1.4 Results Discussion of One-way ANOVA Test

Table 52 details the results of 39 participants who took part in this study and provides some very useful descriptive statistics, including the mean, standard deviation and

95% confidence intervals for the dependent variable (Vulnerability anti-pattern) for each separate group (Formal, Informal), as well as when all groups are combined (Total). The tested conditions of the experiments are based on the type of intervention: “1=Formal training” included 18 participants and 21 participants provided “Informal training”, as well as the mean rank and sum of ranks for the two groups tested. This table is very useful because it indicates that both groups had very similar scores and which group can be considered as having the highest score, which we need to know if we have to interpret a significant result.

Descriptives

Total_Score_Stage2

	N	Mean	Std. Deviation	Std. Error	95% Confidence Interval for Mean		Minimum	Maximum
					Lower Bound	Upper Bound		
Formal	18	11.83	1.654	.390	11.01	12.66	8	14
Informal	21	11.33	2.904	.634	10.01	12.66	0	15
Total	39	11.56	2.393	.383	10.79	12.34	0	15

Table 52 One-way ANOVA test rank

Table 53 shows the output of the ANOVA analysis to determine whether there is a statistically significant difference between two group means. We can see that the significance value is 0.523, p> 0.05. Therefore, the null hypothesis was not rejected that there is no significant difference between “formal” and “informal” interventions in raising awareness of vulnerabilities.

ANOVA

Total_Score_Stage2

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	2.423	1	2.423	.417	.523
Within Groups	215.167	37	5.815		
Total	217.590	38			

Table 53 One-way ANOVA test results

a. Grouping Variable: VulnerabilityAntiPatternCode

b. Not corrected for ties.

To examine the observed difference in stage2 total score between those who received the intervention through “Formal Vulnerability Anti-Pattern” and “Informal

Vulnerability Anti-Pattern", a one-way ANOVA test was performed and found not to be significant. We can conclude from the ANOVA test result that there is no significant difference in the participants mean scores depending on their received intervention type.

7.5 Research Hypothesis

Research Question	Is there a difference in software developers' obtained scores in the stage3 depend on receiving the type of intervention, which performed after the one weeks?	
EH-4	Software developers will manage to retain vulnerability awareness provided by interventions after a gap of one week.	
EH-0	There is no significant difference in software developers' obtained scores in the stage3 after a gap of one week.	
Dependent Variables	Total_Score_stage-2	Total_Score-stage-3
Independent Variables	Vulnerability Anti-Pattern type 1=Formal 2=Informal	
Key Terms	1=Formal 2=Informal	

Table 54 Research question and hypothesis

7.5.1.1 Descriptive Statistical Analysis of Stage_2 and Stage_3 Data

Before performing any formal statistical tests, it was necessary to do a detailed exploration and description of the sample data through descriptive analysis. For the sample data, this was an essential preliminarily step. For that reason, during the descriptive statistical analysis, the sample data had explored thoroughly while including a wide range of useful statistics. For example, exploration of frequencies, mean values and median values. To supplement the descriptive statistics, the graphical representation (Histogram) was included to have a picture of data. The study included the following descriptive statistics:

1. Compare mean, median and mode
2. Frequencies difference of two samples
3. Histograms

1. Compare the Mean, Median and Mode

Table 55 compares the Mean, Median and Mode of total_Score_Stage2 and Total_Score_Stage3, which show that students had managed to retain enough knowledge provided via intervention a week before. However, there is a slight declination of their scores in stage-3 in comparison to stage-2.

Mean, Median and Mode Statistics

		Total_Score_Stage2	Total_Score_stage3
N	Valid	38	31
	Missing	1	8
Mean		**11.8684**	**11.0645**
Median		**12.0000**	**11.0000**
Mode		**12.00**	**11.00[a]**
Std. Deviation		1.47357	2.04834
Variance		2.171	4.196
Range		7.00	7.00
Minimum		8.00	7.00
Maximum		15.00	14.00

Table 55 Descriptive analysis of score of both stages

a. Multiple modes exist. The smallest value is shown

2. Frequency Table

Table 56 analyses the difference in frequencies and measured the central tendency across stage-2 and stage-3 scores (Total_Score_Stage2 and Total_Score_Stage3). Table 56 presents participants frequency with respect to their obtained scores during stage-2, those obtained scores range from 8 to 15. It appears from Table 56 that frequently obtained score by participants during stage-2 is 12 (also known, as central tendency of Total_stage2 was 12). Although, the lowest score was 8 and maximum-score 15.

Total_Score_Stage2

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	8.00	1	2.6	2.6	2.6
	9.00	2	5.1	5.3	7.9
	10.00	2	5.1	5.3	13.2
	11.00	9	23.1	23.7	36.8
	12.00	**11**	**28.2**	**28.9**	**65.8**
	13.00	9	23.1	23.7	89.5
	14.00	3	7.7	7.9	97.4
	15.00	1	2.6	2.6	100.0
	Total	38	97.4	100.0	
Missing		1	2.6		
Total		39	100.0		

Table 56 Stage2 obtained scores frequencies

Table 57 presents scores' frequency obtained during stage3, frequently obtained scores range from 7 to 14. It appears from Table 31 that median in the stage3 was 11 (also known, as central tendency of TOTAL_stage3 was 11). Although, lowest score was14 and highest score 14.

Total_Score_stage3

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	7.00	3	7.7	9.7	9.7
	8.00	1	2.6	3.2	12.9
	9.00	2	5.1	6.5	19.4
	10.00	5	12.8	16.1	35.5
	11.00	**6**	**15.4**	**19.4**	**54.8**
	12.00	5	12.8	16.1	71.0
	13.00	6	15.4	19.4	90.3
	14.00	3	7.7	9.7	100.0
	Total	31	79.5	100.0	
Missing		8	20.5		
Total		39	100.0		

Table 57 Stage3 obtained scores frequencies

3. Histogram

Figures 48 and 49 show the distribution of participants' scores of both stage-2 and stage-3. On the x-axis, the total score of each stage was displayed; on the y-axis, the frequency (numbers of participants) of participants is presented. The graphical presentation of two histograms clearly revealed that both stages highest scores frequency is the almost same, although there is a tendency for participants to score higher in the stage-2: just after receiving the intervention rather than in the stage-3, which was conducted after a week gap. To compare the total score of both stages, it can be observed especially in their mean and standard deviation values those participants' performance declines after a week gap in receiving the intervention.

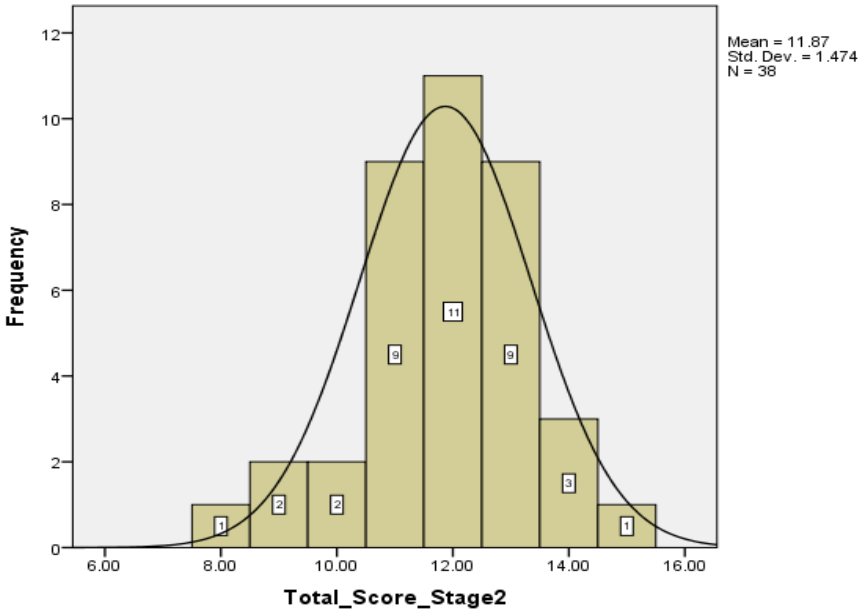


Figure 48 Score distribution of participants in the stage2

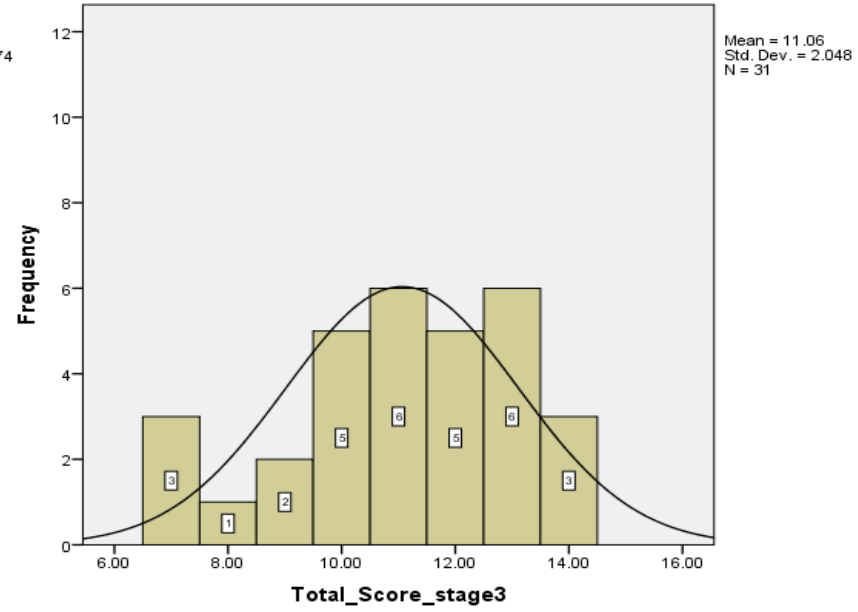


Figure 49 Score distribution of participants in the stage3

7.5.1.2 Discussion

The null hypothesis is not rejected; hence, software developers manage to retain vulnerability awareness provided by interventions after a gap of one week. It can be concluded that in stage-3, the average score is 11 that obtained by 6 participants, although during stage-3 the average score is 12, which was obtained by 11 participants., Therefore, it is clearly justified that during stage-2 the highest obtained score 14 in comparison to stage-3 where the highest obtained score was 15 as shown in Figures 48 and 49. These results suggested that after a gap of 7 days since they received the intervention, participants' ability to identify the root cause of vulnerabilities had reduced slightly. The themes identified in these responses can be compared by the average mean of both stages, which shows a difference in their mean values such as Stage-2=11.87 and Stage-3=11.06 in Table 55. Overall, this is not a significant result.

7.5.1.3 Selection of Appropriate Statistical Test for the Sample Data

Similar to Section 7.3.4, Table 58 shows that data is normalised.

Tests of Normality

		Kolmogorov-Smirnov[a]			Shapiro-Wilk		
	VAP	Statistic	df	Sig.	Statistic	df	Sig.
Total_Score_Stage2	Formal	.225	14	.052	.925	14	.257
	Informal	.247	17	**.007**	.910	17	.099
Total_Score_stage3	Formal	.158	14	.200*	.968	14	.847
	Informal	.173	17	.189	.905	17	.082

*. This is a lower bound of the true significance.

a. Lilliefors Significance Correction

Table 58 Normality tests shown in the 'sig columns'

7.5.1.4 Assumptions of Paired-Sample t Test

Same assumptions followed liked Section 7.3.4.3 to select the following Paired-Sample t Test.

7.5.1.5 Paired-Samples t Test

Tables 59, 60, and 61 present the overall results of the paired samples t Test.

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	Total_Score_Stage2	11.8387	31	1.50769	.27079
	Total_Score_stage3	11.0645	31	2.04834	.36789

Table 59 Both stages paired analysis

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	Total_Score_Stage2 & Total_Score_stage3	31	.295	.107

Table 60 Correlation table

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	Total_Score_Stage2 & Total_Score_stage3	31	.295	.107

Table 61 Correlation table

Paired Samples Test

		Paired Differences				
		95% Confidence Interval of the Difference				
		Upper	t	df	Sig. (2-tailed)	
Pair 1	Total_Score_Stage2 - Total_Score_stage3	1.56493	2.000	30	.055	

Table 62 Paired t test result

7.5.1.6 Result Discussion of Paired-Samples t Test

Table 62 reports the following results: The mean score of the participants of the stage-2 (M= 11.84, SD=1.50) is greater than the mean score of same participants during stage-3 (M= 11.06, SD=2.05) as t= 2.00 and the value of p=.055 is greater than 0.05 as shown in Table 62. However, Table 60 and 61 shown a strong correlation between both stages scores. The two-tailed p>0.05, therefore this is not a significant result.

7.5.2 Assessment of Effectiveness of Intervention Types

7.5.2.1 Research Question

- ✓ *Is there a significant difference between the participants mean scores in the formal intervention group and the informal intervention group after a week gap?*

Case Processing Summary

	Cases					
	Included		Excluded		Total	
	N	Percent	N	Percent	N	Percent
Total_Score_Stage2 * VAP	38	97.4%	1	2.6%	39	100.0%
Total_Score_stage3 * VAP	31	79.5%	8	20.5%	39	100.0%

Table 63 Number of participants

Report

VAP		Total_Score_Stage2	Total_Score_stage3
Formal	Mean	11.8333	10.9286
	N	18	14
	Std. Deviation	1.65387	1.85904
Informal	**Mean**	**11.9000**	**11.1765**
	N	**20**	**17**
	Std. Deviation	**1.33377**	**2.24264**
Total	Mean	11.8684	11.0645
	N	38	31
	Std. Deviation	1.47357	2.04834

Table 64 Both stages mean scores difference

7.5.2.2 Results Discussion

Participants were divided into two groups. One group received the formal intervention, and the other group received the informal intervention. The detail means analysis of dependent variables Total_Score_Stage2 and Total_Score_Stage3 and both stages independent variable VAP (Formal and Informal) has shown in Table 64. It can be concluded that developers have managed to retain more information through the informal intervention (based on the informal Vulnerability Anti-Pattern) rather than formal intervention after a gap of one week.

Report

VAP	Degree		Total_Score_Stage2	Total_Score_stage3
Formal	Computing	Mean	12.0000	10.3333
		N	4	3
		Std. Deviation	1.15470	1.15470
	Gaming	**Mean**	**11.7857**	**11.0909**
		N	14	11
		Std. Deviation	1.80506	2.02260
	Total	Mean	11.8333	10.9286
		N	18	14
		Std. Deviation	1.65387	1.85904
Informal	Computing	Mean	12.6000	10.6000
		N	5	5
		Std. Deviation	1.51658	2.30217
	Gaming	**Mean**	**11.6667**	**11.4167**
		N	15	12
		Std. Deviation	1.23443	2.27470
	Total	Mean	11.9000	11.1765
		N	20	17
		Std. Deviation	1.33377	2.24264
Total	Computing	Mean	12.3333	10.5000
		N	9	8
		Std. Deviation	1.32288	1.85164
	Gaming	Mean	11.7241	11.2609
		N	29	23
		Std. Deviation	1.50941	2.11526
	Total	Mean	11.8684	11.0645
		N	38	31
		Std. Deviation	1.47357	2.04834

Table 65 Means comparison depending on the participants' degree

7.5.2.3 Means Results Discussion

The study sub-divides the participants depending on their degree (C= computing and G=gaming) and type of intervention received as explained in Sections 4.6.1 and 4.6.2. This evaluation gives some interesting outcomes. For example, Table 65 compares both degree students mean scores and concludes that gaming students (G) manage to retain more information in contrast to computing students (C) after a week gap of receiving the intervention.

7.6 Control Experiment

7.6.1 Research Hypotheses

Research Question	Is there a significant difference between the scores of the control and experimental groups?	
EH-5	There is a significant difference between the performance of participants provided with intervention and those not provided with intervention.	
EH-0	There is no significant difference between the performance of participants provided with intervention and those not provided with intervention.	
Depended Variables	Total_Score_stage-1	
Independent Variables	Security Training No= without intervention Yes=with intervention	

Table 66 Control experiment research question and hypothesis

7.6.2 Kruskal-Wallis H test to Compare the Scores of Experimental Group and Control Group

The Kruskal-Wallis H test (sometimes also called the "one-way ANOVA on ranks") is a rank-based non-parametric test that can be used to determine if there are statistically significant differences between two or more groups of an independent variable on a continuous or ordinal dependent variable. It is considered the non-parametric alternative to the one-way ANOVA, and an extension of the Mann-Whitney U test to allow the comparison of more than two independent groups.

7.6.2.1 Stage-1 Control and Experimental Groups Scores Comparison

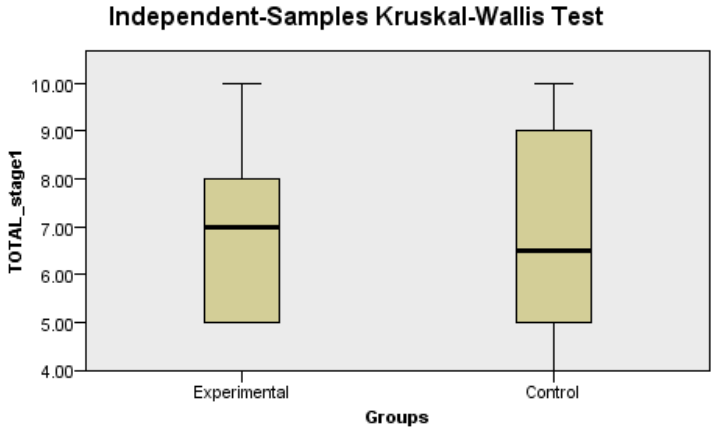
Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of TOTAL_stage1 is the same across categories of Groups.	Independent-Samples Kruskal-Wallis Test	.512	Retain the null hypothesis.

Asymptotic significances are displayed. The significance level is .05.

Table 67 Stage-1 hypothesis test

Table 67 shows that during stage-1 there is no difference between both groups' scores.



Total N	32
Test Statistic	.431
Degrees of Freedom	1
Asymptotic Sig. (2-sided test)	.512

1. The test statistic is adjusted for ties.
2. Multiple comparisons are not performed because the overall test does not show significant differences across samples.

Figure 50 Stage-1 control and experimental groups scores comparison

As shown in Figure 50, the Kruskal-Wallis test could not provide strong evidence of a difference ($p > 0.05$) between the mean ranks of the groups. Therefore, the null hypothesis is not rejected because there is no significant difference between the performance of participants provided with intervention and those not provided with intervention.

7.6.2.2 Stage-2 Control and Experimental Groups Scores Comparison

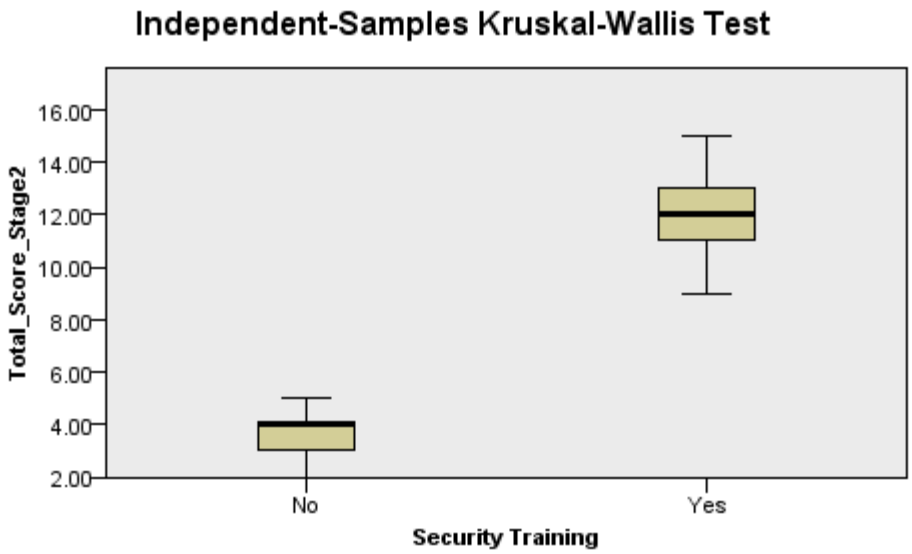
Hypothesis Test Summary

	Null Hypothesis	Test	Sig.	Decision
1	The distribution of Total_Score_Stage2 is the same across categories of Security Training.	Independent-Samples Kruskal-Wallis Test	.000	Reject the null hypothesis.

Asymptotic significances are displayed. The significance level is .05.

Table 68 Stage-2 hypothesis test

Table 68 shows that during stage-2 there is a difference in scores of experimental and control groups.



Total N	52
Test Statistic	30.771
Degrees of Freedom	1
Asymptotic Sig. (2-sided test)	.000

1. The test statistic is adjusted for ties.
2. Multiple comparisons are not performed because there are less than three test fields.

Figure 51 Stage-2 control and experimental groups scores comparison

Figure 51 shows that the Kruskal-Wallis test provided very strong evidence of a difference ($p < 0.001$) between the mean ranks of stage-2 scores of trained and non-trained groups. Therefore, the null hypothesis is rejected that there is no significant difference between the performance of participants provided with intervention and those not provided with intervention.

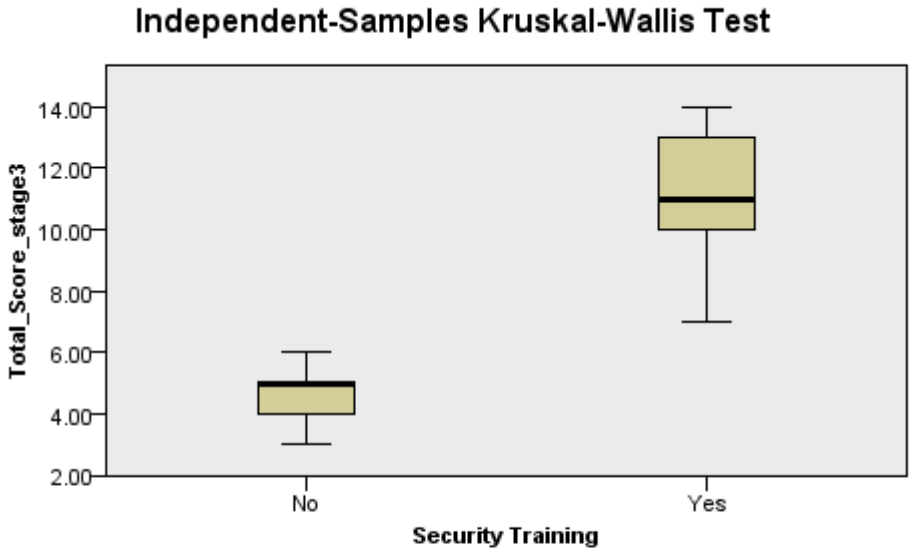
7.6.2.3 Stage-3 Control and Experimental Groups Scores Comparison

Hypothesis Test Summary			
Null Hypothesis	**Test**	**Sig.**	**Decision**

1	The distribution of Total_Score_stage3 is the same across categories of Security Training.	Independent-Samples Kruskal-Wallis Test	.000	Reject the null hypothesis.
Asymptotic significances are displayed. The significance level is .05.				

Table 69 Stage-3 hypothesis test

Table 69 shows that during stage-3 there is a difference in scores of experimental and control groups.



Total N	45
Test Statistic	28.646
Degrees of Freedom	1
Asymptotic Sig. (2-sided test)	.000

1. The test statistic is adjusted for ties.
2. Multiple comparisons are not performed because there are less than three test fields.

Figure 52 Stage-3 control and experimental groups scores comparison

Figure 52 shows that the Kruskal-Wallis test provided very strong evidence of a difference ($p < 0.001$) between the mean ranks of stage-2 scores of trained and non-trained groups. Therefore, the null hypothesis is rejected that there is no significant difference between the performance of participants provided with intervention and those not provided with intervention. However, it does not reflect which group performed better.

7.6.2.4 Conclusion

The outcome of the Kruskal–Wallis post hoc test indicates that during stage-1 there are no differences between group scores. However, during stage-2 and stage-3 there is a significant difference between group scores. These differences are not significant. In order to determine which group is significantly better than other, a Mann-Whitney U test was used.

7.6.3 Mann-Whitney U Test to Compare the Scores of Experimental Group and Control Group

The Mann-Whitney U test was used to compare differences between independent groups, such as control and experimental groups. The Mann-Whitney U test compares the differences between groups scores to determine if there are differences between groups.

7.6.3.1 Stage-1 Scores Comparison: Mann-Whitney U Test Ranks Table

Table 70 provides information regarding the output of the Mann-Whitney U test. It shows the mean rank and sum of ranks for the two groups tested: experimental and control.

Ranks				
	Security Training	N	Mean Rank	Sum of Ranks
TOTAL_stage1	No	14	15.75	220.50
	Yes	18	17.08	307.50
	Total	32		

Table 70 Stage-1 Mann-Whitney U test ranks

The mean rank indicates which group performed better, overall; namely, the group with the highest mean rank. Table 70 indicates that the experimental group obtained higher scores in comparison to the control group.

7.6.3.1.1 Mann-Whitney U Test Table

Table 71 shows the significant value of the test. Specifically, Table 71 provides the U statistic, as well as the asymptotic significance (2-tailed) p-value.

Mann-Whitney U Test Statistics[a]	
	TOTAL_stage1
Mann-Whitney U	115.500

Wilcoxon W	220.500
Z	-.408
Asymp. Sig. (2-tailed)	.684
Exact Sig. [2*(1-tailed Sig.)]	.694[b]

Table 71 Stage-1 Mann-Whitney U statistics outcome

a. Grouping Variable: Security Training

b. Not corrected for ties.

From this data, it can be concluded that scores of the experimental group were not statistically significantly higher than the control group (U =115.50, p >0.05). There is no significant difference between experimental and control groups, which shows both groups obtained the same scores not receiving the intervention based on vulnerability anti-pattern.

7.6.3.2 Stage-2 Scores Comparison: Mann-Whitney U Test Ranks Table

Table 72 provides information regarding the output of the Mann-Whitney U test. It shows mean rank and sum of ranks for the two groups tested: without security training, and with security training

Ranks				
	Security Training	N	Mean Rank	Sum of Ranks
Total_Score_Stage2	No	14	7.50	105.00
	Yes	38	33.50	1273.00
	Total	52		

Table 72 Stage-2 Mann-Whitney U test ranks

Table 72 shows the highest mean rank of the trained group, which indicates that the trained group performed better as having higher scores. In this case, the participants with security training had the highest scores.

7.6.3.2.1 Mann-Whitney U Test Table

From Table 73 data, it can be concluded that scores of the trained group were significantly higher than the non-trained group (U =.000, p <0.001). There is a significant difference between the performance of participants provided with intervention, as compared to those not provided with training.

Mann-Whitney U Test Statistics[a]	
	Total_Score_Stage2
Mann-Whitney U	.000

Wilcoxon W	105.000
Z	-5.547
Asymp. Sig. (2-tailed)	.000

Table 73 Stage-2 Mann-Whitney U statistics outcome

a. Grouping Variable: Security Training

7.6.3.3 Stage-3 Scores Comparison: Mann-Whitney U Test Ranks Table

Table 74 provides information regarding the output of the Mann-Whitney U test. It shows mean rank and sum of ranks for the two groups tested: without security training, with security training.

Ranks				
	Security Training	N	Mean Rank	Sum of Ranks
Total_Score_stage3	No	14	7.50	105.00
	Yes	31	30.00	930.00
	Total	45		

Table 74 Stage-3 Mann-Whitney U test ranks

Table 74 shows the highest mean rank of the trained group, which indicates that the group with security training performed better. In this case, the participants with security training had the highest scores.

7.6.3.3.1 Mann-Whitney U Test Table

From Table 75, it can be concluded that scores of the trained group were significantly higher than the non-trained group (U =.000, p <0.001). There is a significant difference between the performance of participants provided with intervention, and those not provided with intervention.

Mann-Whitney U Test Statistics[a]	
	Total_Score_stage3
Mann-Whitney U	.000
Wilcoxon W	105.000
Z	-5.352
Asymp. Sig. (2-tailed)	.000

Table 75 Stage-3 Mann-Whitney U statistics outcome

a. Grouping Variable: Security Training

7.7 Pilot-Study-II Overall Results Summary

The original hypothesis was formulated in order to determine whether participants were able to identify vulnerabilities with interventions based on anti-patterns as knowledge transfer mediums. The experimental groups score statistical analysis shows that developers improve their awareness of poor security practices (vulnerabilities) using VAPs. The research designed two different types of anti-patterns ("Informal" and "Formal" Section 4.6). To verify the efficacy of interventions, the experimental group was divided into halves and provided different interventions. The results indicate that there is no significant difference between "Informal" and "Formal" Vulnerability Anti-Pattern in term of retaining the vulnerability knowledge. The most interesting finding was that an "informal intervention was more effective in helping participants to retain an awareness of vulnerabilities after a one week gap in comparison to the formal intervention.

A control experiment was carried out to determine if there were any significant differences between the performance of participants provided with intervention and those not provided with an intervention. A Quasi-experiment was chosen because the students could not be chosen by chance. The results of the experiment rejected the null hypothesis: There is no significant difference between the performance of participants provided with the intervention and those not provided with the intervention.

It is evidenced by the statistical analysis of pilot-study-II results (see Table 78) that developers improve their awareness of poor security practices (vulnerabilities). However, there is no significant difference between "Informal" and "Formal" Vulnerability Anti-Pattern. The most interesting finding was that an "informal intervention was more effective in order to help participants to retain an awareness of vulnerabilities after a one week gap in comparison to a formal intervention. Table 76 presents the pilot-study-II results summary.

Questions	Test name	p-value	Normality distribution
Does intervention, based on the use of "formal or informal Vulnerability Anti-Pattern", improve participants'	Paired-Samples T Test	P<0.001	Yes

ability to identify the root-causes of vulnerabilities?			
Is there a difference between "formal" and "informal" intervention in order to provide information of vulnerabilities?	Mann-Whitney Test	P>0.05	No
Is there a difference in the scores obtained by software developers in stage3 depending on which type of intervention they received after a week gap?	Paired-Samples T Test	P>0.05	Yes
Is there a significant difference between the scores of the control and experimental groups?	Mann-Whitney U Test	p <0.001	No

Table 76 Pilot-Study-II results summary

8 Industrial Study (Qualitative Approach)

8.1 Introduction

This chapter details the results of the experimental study which was performed with professional software developers to evaluate the efficacy of the vulnerability anti-patterns in improving awareness of vulnerabilities. The industrial study is divided into three parts

- Section 8.2 explains the experiment design and questionnaires structure which followed the same experimental design as the pilot-study-II.
- Section 8.3 summarises the results from this study.
- Section 8.4 empirically evaluates the results based on the qualitative approach to prove/ disapprove the hypotheses of this thesis as mentioned in Section 1.7.

8.1.1 General Description

The Industrial study's key objectives were to evaluate professional software engineers' (software developers) understanding of security flaws that lead to vulnerabilities and to measure the effectiveness of Vulnerability Anti-Patterns to help developers in improving their understanding of vulnerabilities.

8.2 Method

8.2.1 Experiment Study Description

There is no established method to evaluate Vulnerability Anti-Patterns (VAP) in order to provide essential awareness of vulnerabilities. Evaluating and measuring Vulnerability Anti-Pattern effectiveness with a limited number of participants is fraught with difficulty and potential bias. Due to the small sample size, it is difficult to apply a quantitative method to perform statistical evaluation such as that applied during the Pilot study-II. Pilot study II shows an improvement in developers' awareness about vulnerabilities via VAPs. Therefore, a qualitative approach for evaluating the effectiveness of VAP is adopted here. The purpose of this approach is to demonstrate the efficacy of VAP in aiding developers in finding the vulnerabilities within the vulnerable code/UML diagram, thus determining its overall effectiveness in a professional environment.

To eliminate bias in the measurement of effectiveness, the five stage experiment study was designed (see Section 8.2.2 design of experiment study). The design of the experiment was discussed earlier in the Pilot study-II. The experiment was developed independently to avoid any bias towards a particular technique, domain or environmental requirement. Instead, it was designed to illustrate the many ways that intervention could be measured and evaluated. This ensures the evaluation framework is fit for general purpose and reusable as a common method of evaluation.

8.2.2 Experiment Design Structure

In the qualitative process of evaluation, the experimental study was carried out alongside the semi-structured interviews as shown in Table 77.

The semi-structured interview was conducted with participants to overcome the small sample size and time limitations, which provides in-depth analysis with feedback on VAP. A suitable method of evaluation in a small sample size is to do a detailed assessment (Newton 2010).

The study comprised of five stages supported by an intervention via Vulnerability Anti-Patterns. The engineers completed a questionnaire, which consist of 15 questions relating to five vulnerabilities.

		Group A	Group B
Stage-1	Pre-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate participants' ability to measure their awareness about vulnerabilities	
Stage-2	Intervention Session		
	Input	Group A-Formal Vulnerability Anti-Pattern	Group B-Informal Vulnerability Anti-Pattern
	Output	Provide information of vulnerabilities	
Stage-3	Post-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate participants' ability to measure improvement/ deterioration after intervention provided by VAPs	
Stage-4	Semi-structured interview with Participants (Only with industry)		
After 1 Week (one week gap)			
Stage-5	Post-Post-Assessment Survey Study		
	Input	Questionnaire comprised of vulnerable codes or UML diagram	
	Output	Evaluate results to measure how much participants able to retain the information from the provided intervention based on VAPs after a gap of one week.	

Table 77 Description of experiment study structure, including all stages inputs and outputs description.

8.2.3 Experiment Questions' Structure

Similar to Section 7.2.3, which was essentially constructed to test engineers' understanding and awareness of vulnerabilities while specifying their root-causes during the Software Development Lifecycle (SDLC) phases from where the vulnerability originated. The aim is to categorise flaws based on SDLC:

requirement specification phase flaw, design phase flaw and implementation phase flaw. Subsequently, this directs developers' attention towards the SDLC phase from where the vulnerability initiated.

Each question was designed to ask software developers about the following necessary information: For example, part-1 of each question assessed participants' actual knowledge about the vulnerability while providing vulnerable code or UML diagram, part-2 investigated participants' understanding about misused or exploitation, and part-3 inspected participants' ability to recognise and classify the vulnerabilities using the terminology of the cybersecurity community. The question structure is as follows:

1. Part-1: Vulnerable code or UML diagram
2. Part-2: Misused or exploited explanation
3. Part-3: Identify vulnerability formal name

8.2.4 Vulnerability Sample Size

For Java developers, the included vulnerabilities are:

1) Missing Authentication,
2) Missing Authorization,
3) SQL injection
4) Buffer Overflow
5) Integer Overflow

For C# developers, the included vulnerabilities are:

1) Missing Authentication
2) OS Command Injection
3) Cross-Site Scripting
4) SQL Injection
5) Cross-site Request Forgery

8.2.5 Participants Sample Size

The study was conducted with five professional software engineers. Table 78 shows the demographics of the participants.

Participants Assign Names	Degree	Software Development Experience
P1	Ethical Hacking	2 years

P2. P3, P4	Computing	2 years, 1 year, 2 years
P5	Computer and Electric Engineering	5 years

Table 78 Participants information

8.3 Results

8.3.1 Research Question

✓ *Does the Vulnerability Anti-Pattern (VAP) improve developers' understanding (awareness) to identify the root-cause of vulnerabilities in order to help developers to the creation of secure software systems?*

To investigate this question, the experimental study was performed with professional software engineers. During the experiment study, each participant was provided with an intervention based on Vulnerability Anti-Pattern and evaluated. The developers obtained scores before and after receiving the invention.

The evaluation experiment study was developed with the purpose of fairly assessing the ability of VAP to provide a useful understanding of vulnerabilities. The experiment study is structured to determine the followings.

1. Intervention helps participants' ability to identify the root-cause of Vulnerabilities.
2. Intervention helps participants' ability to recognise and classify vulnerabilities using the terminology of the cybersecurity community.
3. There is no difference between "Informal" and "Formal" interventions.

8.3.2 Intervention helps participants' ability to identify the root-cause of Vulnerabilities

✓ *The intervention study, which is based on the Vulnerability Anti-pattern will improve participants' ability to identify the root-cause of vulnerabilities during the software development process.*

In Pilot-Study-II Section 7.2, it was proved while using quantitative statistical analysis that intervention is useful in order to improve participants' ability to identify the root-cause of vulnerability. However, this study is based on qualitative analysis. The industrial study designed in two programming languages based on engineers' awareness:

1) **For Java Engineers**

- **Stage-1: Pre-Assessment**
  Out of 5 questions, Java based engineers only managed to answer 2 questions correctly.
- **Stage-3: Post –Assessment**
  After getting the intervention, out of 5 questions engineers managed to answer 4 questions correctly.
- **Stage-5: Post-Post-Assessment**
  Similarly, to stage-3, after a gap of one week in intervention, engineers answered 4 questions correct out of 5.

2) **For C# Engineers**

- **Stage-1: pre-Assessment**
  Out of 5 questions, C# engineers answered the 3 questions correct.
- **Stage-3: Post –Assessment**
  After getting the intervention, all engineers managed to answer 5 questions correct.
- **Stage-5: Post-Post-Assessment**
  Unlike stage-3, after a gap of one week in intervention, engineers managed to answer 4 questions correct out of 5.

8.3.3 Results Discussion

Identifying vulnerabilities within code samples or UML diagram is a complicated process. Those engineers who closely worked with the security of software systems such as P3, P5, easily target the vulnerabilities in the code samples. However, engineers with a computing background struggle to find the vulnerabilities in the code samples such as P1, P2.

Stage-1 and Stage-3 scores comparison presented in Table 79, which clearly demonstrates the improvement in engineers' scores after receiving the intervention. Furthermore, those engineers (i.e. P1, P2, and P4), who find difficulty in identifying the vulnerabilities during stage-1, after receiving the intervention, they also performed well and improved their scores during Stage-3.

However, engineers' background knowledge also has an impact on their ability to find the vulnerability in the code sample such as P2 engineer with computing background scored less than P3 engineer (P3) with cybersecurity

background. After receiving the intervention, both participants scored well during Stage-3. Therefore, overall results suggest that intervention improves participants' ability to identify the root-cause of vulnerabilities during the software development process.

Participants	Java	C#	Before Intervention	After Intervention
			Stage-1 score	Stage-3 score
P1	Y		6	12
P2	Y		6	14
P3		Y	11	14
P4		Y	10	13
P5		Y	11	13

Table 79 Participants' scores before and after intervention

8.3.4 Intervention helps participants' ability to recognise and classify vulnerabilities using the terminology of the cybersecurity community

✓ *The intervention, which is based on Vulnerability Anti-pattern, will improve participants' ability to recognise and classify vulnerabilities using the terminology of the cybersecurity community.*

The industrial study was designed in two programming languages based on engineers' awareness:

1) **For Java Engineers**

- **Stage-1: Pre-Assessment**
  Out of 5 questions, Java engineers managed to answer 4 questions correctly.
- **Stage-3: Post –Assessment**
  After getting the intervention, out of 5 questions engineers managed to answer 4 questions correctly.
- **Stage-5: Post-Post-Assessment**
  After a gap of one week of receiving the intervention, engineers answered all questions correctly.

2) **For C# Engineers**

- **Stage-1: Pre-Assessment**
  Out of 5 questions, C# engineers answered all question correctly.
- **Stage-3: Post –Assessment**
  After receiving the intervention, all engineers managed to answer all questions correctly.
- **Stage-5: Post-Post-Assessment**
  Similarly, to stage-3, after a gap of one week in having the intervention, engineers managed to answer all question correctly.

8.3.5 Results Discussion

Awareness of cybersecurity terminology is an essential aspect in order to understand vulnerabilities. During this part of the experiment, the study results show that all engineers are aware of the vulnerabilities' terminologies that are used by the cybersecurity community.

Table 80 compares the participants' score in order to answer the part-3 of the questionnaire, which inquires the participants' awareness of software flaws called by cybersecurity experts.

Two engineers were unaware of Cross-Site Request Forgery and missing authorisation vulnerabilities (flaws) terminologies during Stage-1 such as P1 and P2. The reason underlying was caused by their expertise and knowledge being derived elsewhere. For example, web-based developers have a different set of security priorities than the developers who worked on desktop applications. During Stage-1, engineers struggled to find the Cross-Site Request Forgery vulnerability, but with the support of intervention, they understood this vulnerability and managed to get it correct during Stage-3. However, some engineers lack the proper understanding of some vulnerabilities terminologies; for example, they got confused between information leakages with missing authorisation terminologies. As both vulnerabilities are interdependent/ interlinked to each other, thus we considered both answers correct in this case.

Participants	Java	C#	Before Intervention	After intervention	
			Stage-1 score	Stage-3 score	Stage-5 score
P1	Y		4	5	5
P2	Y		3	4	5
P3		Y	5	5	5
P4		Y	5	5	5
P5		Y	5	5	5

Table 80 Participants scores to identify vulnerabilities terminologies

8.3.6 There is no difference between "Informal" and "Formal intervention"

✓ *There is no difference between "Informal" and "Formal" intervention in order to provide information about vulnerabilities.*

As mentioned previously in Chapter 4 Section 4.6. VAPs were designed in two types: Informal and Formal. Engineers were provided with two kinds of interventions. The experiment study aims to measure which kind of intervention is more effective.

1) **Informal Intervention**

- **Stage-3: Post –Assessment**
  Out of 5 participants, 3 were provided with informal intervention, who scored 12, 13, and 14 as shown in Table 81.
- **Stage-5: Post-Post-Assessment**
  After a gap of one week of receiving an informal intervention, out of 3 engineers, only 1 performed better while others were managed to retain their scores as compared to Stage-3.

2) **Formal Intervention**

- **Stage-3: Post –Assessment**
  After receiving a formal intervention, engineers scored 14 and 13 respectively as shown in Table 81.
- **Stage-5: Post-Post-Assessment**
  After a gap of one week, the formal intervention had a mixed response such as P2 performed better; however, P5 performance was declined as compared to Stage-3.

8.3.7 Results Discussion

Overall, the results suggest that both interventions help participants to improve their ability to identify and recognise vulnerabilities, although the small sample size is a limiting factor. The null hypothesis cannot be rejected due to the sample size, so there is no difference between "informal" and "formal" intervention.

As shown in Table 81, it is noticeable that the Informal intervention appears easy for engineers' understanding which is confirmed through participant P2 score of 14 during Stage-3 and scored 15 during Stage-5. Although after

receiving a formal intervention, participant P5 scored 13 during Stage-3 and declined score (12) during Stage-5.

As discussed previously in Section 4.4, there is no current methodology to evaluate VAP types; thus, this study concluded that both “Informal” and “Formal” interventions are equally useful in order to provide information of vulnerabilities.

Participants	Informal Intervention		Formal Intervention	
	Stage-3 score	Stage-5 score	Stage-3 score	Stage-5 score
P1	12	13		
P2			14	15
P3	14	14		
P4	13	13		
P5			13	12

Table 81 Participants scores and intervention type

8.4 Discussion of Overall Results Including Semi-Structure Interview Data

This section analyses the questionnaires score and semi-structured interview data of participants which concludes in two stages.

8.4.1.1 Pre-Assessment Stage

The first concern during the pre-assessment stage was whether engineers have an understating of vulnerabilities, do they know how vulnerable codes lead to a vulnerability. Interestingly, the results obtained from the pre-assessment stage shows that engineers had a partial understanding of vulnerabilities.

Experiment study Q-1 (see appendices Table 67) relates to the create Bank Account sub-system, which was comprised of a figure and UML diagram. This was used to evaluate the understanding of the engineers about Missing Authentication vulnerability.

This question is based on the UML diagram, which was appeared complex for engineers. Overall, during Stage-1, out of 5 engineers only 2 were managed to answer correctly. Following are the reasons to use the UML diagram to evaluate Missing Authentication vulnerability.

1) Try to map vulnerability within Software Development Lifecycle. As targeted vulnerability commonly raised flaw during the design phase, so

we used UML diagram to assess engineers' understanding about from where and when Missing Authentication vulnerability has occurred.

2) Evaluate and test engineers' awareness of the root cause of vulnerability.

Engineers faced difficulty in answering this question due to a lack of background knowledge and lack of UML diagram usage in their routine. As participants said during the informal semi-structured interview,

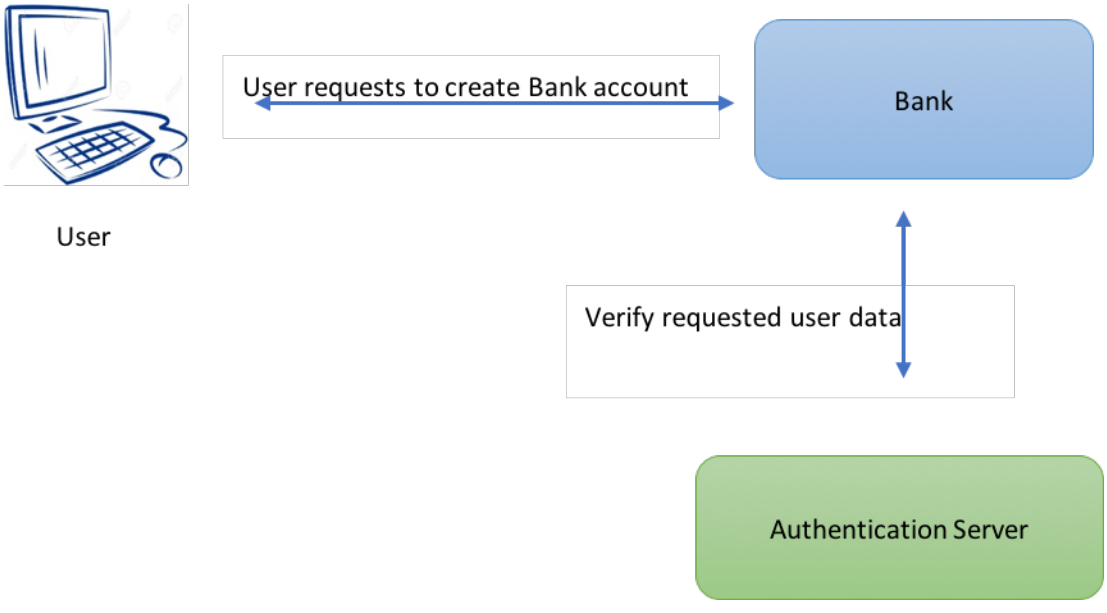
P1 said: "Especially in the UML for the authentication authorised one I got a bit confused because may I am not used UML diagrams anymore".

P3 said:" I say that UML diagram in Q-1 is not effective just simply not explicitly and everything it is saying".

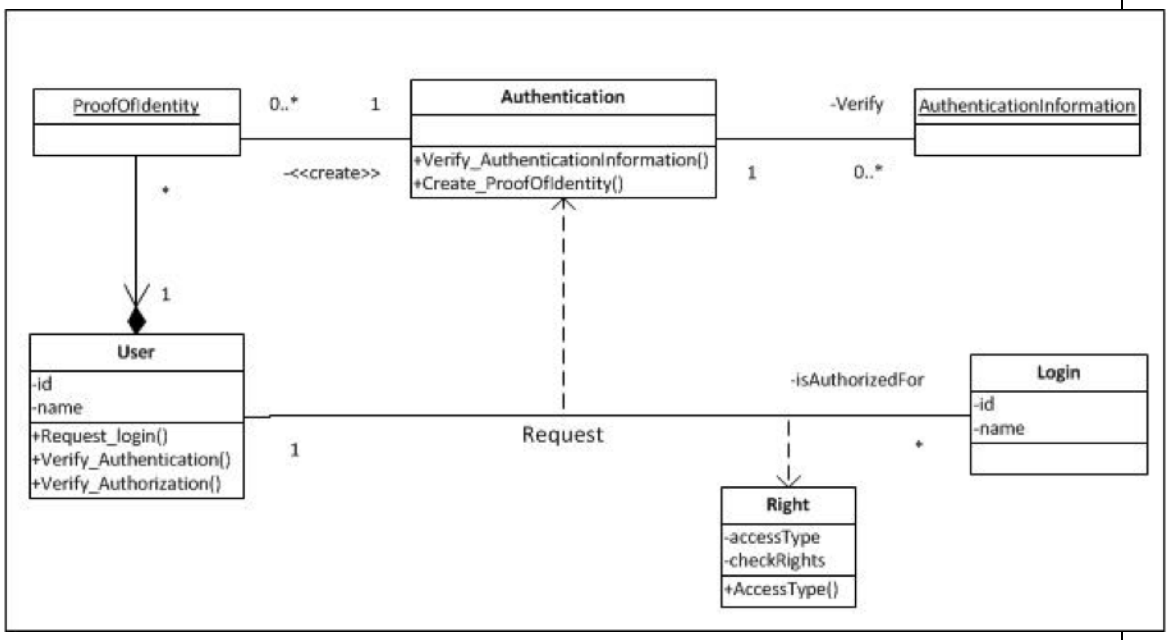
Only in this particular case, it can be concluded that engineers do not use UML diagrams; that's why they could not answer the Q-1 related to Missing Authentication vulnerability during Stage-1 as shown in Table 82.

Overall, during pre-assessment-stage, all engineers have managed to answer Buffer Overflow and SQL injection vulnerability related questions correctly.

Consider the scenario in Figure below that is based on the design of a bank login system, which creates an account for the valid users only.



The UML class diagram below demonstrates the createBankAccount sub-system of the online banking system. As createBankAccount is a critical function, can you verify the authentication mechanism to ensure that the user has the permission to create a new bank account (a bank account is a critical object)? On a scale of 1 (not secure) to 3 (very secure), how secure is this authentication mechanism?



Choose one of the following answers

- Not secure
- Average
- Very secure
- Other:

Table 82 Missing authentication question

8.4.1.2 Post-assessment & Post-Post-Assessment Stages

After receiving an intervention during post-assessment stages, there is a significant improvement in all engineers' scores.

Engineers find the intervention an effective way to provide information of security flaws in the vulnerable code. As participants said during the informal semi-structured interview,

> *P3 said: "Yes, VAP was very clear, certainly with the code examples, definitely make it clear and make it standout".*

The main concern is related to the effectiveness of VAP in terms of whether it provides enough information about a vulnerability.

P2 said: "the document (VAP) we received just looked like descriptions of security vulnerabilities".

P1 said: "In reading the anti-pattern, the sheet you gave us was more helpful, I thought".

P4 mentioned: "apart, looking at code reviewing code, for example, the ability to identify those is, yea is certainly useful is".

The second trend was the improvement of engineers understanding in the part-2 of each question that was related to the exploitation of vulnerable code.

For example,

P2 said: "VAPs did give me some insight into security vulnerabilities".

P5 said: "but I think it would probably be a good reference guide".

Furthermore,

P3 said: "VAPs are a good way to explain the vulnerability in itself, so developers might understand the idea, especially inexperienced developers who might not understand why if they are putting their code would be a vulnerability rather than from the opposite direction understanding how something would be exploited immediately. I think it could be a very effective way of preventing vulnerabilities".

The third trend was how user-friendly VAPs are for engineers. The vulnerable code example in the VAP support engineers to understand the vulnerability in a well-defined fashion such as

P5 said: "I was never got across to cross site forgery vulnerability whatever it called, so VAP of cross-site request forgery example was a very good example, because I thought, ah right that makes sense".

P4: "certainty in CRSFG one there, the example you gave in the sheet certainly very clear, you know, when you just picturing in your head exactly how it works, so the description and explanation are good for us".

P5 said:" one thing is a probably good idea in VAPs because your kind understands the general idea of it, but you see the code examples probably quite helpful particular for the cross-site request forgery.so I think having a code example that in particular, one is good".

In particular, engineers find VAPs more helpful against those vulnerabilities in which they do not have background knowledge.

P5 said: "I think they are pretty understandable, I think is, well at least for programmers, you, I think you beat them understand, why you won't do this, what wrong with it, and how to avoid it".

Some important comments raised by participants are:

P4 said: "They (VAPs) would be useful only if readily available and referred to regularly. Would also be useful for code reviews, but again only if readily available/easily accessible".

P3 said: "be exploited immediately. I think it could be a very effective way of preventing mistakes made of being in the first place".

Overall these stages results suggest that engineers easily understand VAPs and their usage helps engineers to identify and recognise vulnerabilities effectively.

The third trend was how user-friendly VAPs are for engineers. The vulnerable code example in the VAP supports engineers in understanding the vulnerability in a well-defined fashion such as Cross-Site Request Forgery vulnerability.

8.4.1.3 Industrial Study Overall Results Summary

The sample size of the industrial study was very small in comparison to PS-I and PS-II. Due to this reason, it is difficult to draw a significant conclusion. However, overall study results show essential outcomes such as intervention based on Vulnerability Anti-Pattern (VAP) is an effective way to provide developers with a necessary awareness of poor security practices that cause vulnerabilities. This knowledge transformation can bridge the security knowledge gap between software developers and cybersecurity experts in order to the creation of secure software systems.

The study used a qualitative research method, including semi-structured interviews, which analysed experiment results to investigate developers' understandings of recurrent vulnerabilities and to the measure effectiveness of VAPs in order to improve vulnerabilities awareness. Several interesting trends emerged from this evaluation study to show the potential of VAPs as a solution against recurrent vulnerabilities to prevent and mitigate them. Industrial study overall results summary is:

- For all participants, intervention improves their ability to identify the root-cause of Vulnerabilities.
- In general, participants were aware of vulnerabilities using the terminology of the cybersecurity community; however, some of them were unaware due to their background knowledge.
- Due to small sample size, there is no significant difference between "Informal" and "Formal intervention".

9 Discussion

This chapter will discuss the results, which were obtained and analysed in pilot-study-I, pilot-study-II and the industrial study reported in Chapters 6, 7, and 8, respectively.

Overall, the discussion will examine how the results relate to the research question “Can a pattern-based approach (Vulnerability Anti-Pattern) be effective in bridging the security knowledge gap between software developers and security experts in order to help developers in the creation of secure software systems?”

9.1 Discussion of Results

9.1.1 Reflection on Pilot-Study-I

Pilot-study-I aimed to evaluate the students’ understanding of recurrent vulnerabilities. There was no intervention during this study. Overall, the Pilot-study-I results suggest that computing degree related students (Developers) were lacking an effective awareness, which would enable them to identify recurring security flaws, coupled with an awareness of how malicious hackers can exploit these flaws

However, background knowledge has a significant impact on developers’ abilities to prevent and mitigate vulnerabilities. For example, Ethical Hacking degree students performed well in identifying vulnerabilities in comparison to computing degree students. Furthermore, statistically significant results gained while evaluating the questionnaire results of participants shows that computing degree students lack the vulnerability awareness in comparison to ethical hacking degree students. However, computing students perform better than ethical hackers in coding phase related vulnerabilities such as buffer overflow and integer overflow. The primary goal of PS-I was to investigate the participants’ (students’) ability to identify software development errors that lead to vulnerabilities. Participants were given vulnerable code samples or UML class diagram and asked them to identify development errors. Furthermore, to explore the participants’ ability to know how malicious hacker exploit these vulnerabilities. The PS-I concluded that software developers are different from ethical hackers due to their background knowledge and type of tasks. Software developers worked to develop software, but ethical hackers worked as penetration testers to

find security flaws or vulnerabilities. The groups: penetration testers and software developers did not share common ground knowledge. The results of this case study, therefore, raise the concern that computing students lack the awareness of vulnerabilities and do not know how malicious hackers can exploit these. This case study also highlighted another fact that ethical hacking students lack the understating of coding level vulnerabilities.

9.1.2 Reflection on Pilot-Study-II

Based on the findings and limitations of the pilot-study-I, as discussed in Section 6.3.6, which raised some concerns and justified further investigation in PS-II such as vulnerable code samples would need to understandable and accessible to explain the target vulnerability, as discussed in following chapter PS-II. The experiment design of PS-II considered the PS-I limitations and designed a more structured experiment. The participants were split up into two distinct groups:

1) Experimental group (with the invention based on Anti-patterns)
2) Control group (without intervention)

During PS-II, the guideline and examples were chosen with great care.

The experiment was designed to test the following hypothesis

H1: There is a significant difference between the performances of those students who received the intervention and those who did not receive the intervention.

The null hypothesis is stated as

H0: there is no significant difference between the performance of the students with or without intervention.

The results of the variance analysis indicate that performance of students provided with the intervention based on Anti-patterns was significantly better than those not provided with intervention. This provides quantitative support to our argument that Anti-patterns have a positive impact on transferring knowledge of poor development practices. The results from our quantitative analysis were confirmed by an observed difference in scores between the members of two groups during PS-II. The experimental group of students were demonstrated remarkable improvement in finding the vulnerabilities in code once they were provided with the VAP patterns support as an intervention.

It is evident from the statistical analysis of pilot-study-II results that developers improved their awareness of poor security practices, after receiving the intervention based on Vulnerability Anti-Pattern. Furthermore, statistically significant results showed, that intervention helped participants' developers to understand vulnerabilities with the insight of how they can prevent them. Furthermore, just after receiving the intervention (without a week gap), there is no statistically significant difference between the "Informal" and "Formal" interventions. However, after a week, there was a significant trend of showing that "informal intervention is more effective in order to retain vulnerabilities awareness".

9.1.3 Reflection on Industrial Study

In comparison to PS-I and PS-II, the industrial study was based on qualitative analysis due to the very small sample size. The results of the industrial study were unable to provide statistical significance trends, concluded that with the use of VAPs professional engineers (developers) improved their ability to identify the root-cause of vulnerabilities. In general, participants were aware of vulnerabilities' terminologies used by the cybersecurity community such as cross-site request forgery and potential dangerous function calls. In addition, two engineers lacked information of integer overflow and cross-site request forgery vulnerabilities, which could be due to their background knowledge. These results are not significant as the sample size was too small, and engineers had different levels of experience with different background knowledge. Furthermore, the study was unable to show the difference between "Informal" and "Formal" interventions. Several interesting trends emerged from this evaluation study to show the potential of VAPs as a knowledge transfer mechanism.

9.1.4 Conclusion

As discussed in Chapter 3, the complicated structure of vulnerability databases (VDBs) and their inadequacies in capturing and transferring vulnerability knowledge via existing pattern-based approaches pose significant challenges to software developers. The contention between raising vulnerability awareness and the transfer of security knowledge is paramount to success in an effective way of capturing and transferring of vulnerability knowledge, in comparison to the

efficacy existing efforts such as security patterns, attack patterns and software fault patterns. As discussed in Section 2.11.1 due to a lack of shared understanding between cybersecurity and software engineering communities – there is a distinct communication gap when it comes to addressing recurring security problems, known as vulnerabilities. This research designed a new approach Vulnerability Anti-Pattern (VAP), which includes two key components: (i) an **anti-pattern** which captures and transforms information on vulnerability and its common footprints to explain how a malicious hacker can exploit, and (ii) a **pattern** which provides solutions and mitigation techniques against the vulnerability. Moreover, as developers are familiar with pattern-based approaches, Vulnerability Anti-Patterns are designed to provide understandable vulnerability knowledge to developers. To evaluate the efficacy of VAP, empirical studies were conducted to determine whether VAPs are significantly effective to provide vulnerabilities awareness to developers as a knowledge transfer medium. The results of PS-I, PS-II and the industrial study confirmed the effectiveness of VAPs to raise awareness of vulnerabilities. These experiments quantitatively and qualitatively confirmed earlier assertions about the usefulness of VAPs of presenting poor security practices via Anti-patterns. The results of PS-II provided significant outcomes to improve participants vulnerability awareness via VAPs. These differences in participants' performance can be attributed to the effectiveness of VAPs in presenting vulnerability knowledge as discussed in Chapter 4.

Background knowledge of the participants was another impacting factor, which suggested that software developers are different from ethical hackers due to their background knowledge and nature of work. Software developers worked to develop software; however, ethical hackers worked as penetration testers in order to find security flaws or vulnerabilities. Both groups: penetration testers and software developers do not share common ground knowledge of software exploitation. An industrial study was performed with professional software developers and also confirmed the potential of the vulnerability anti-patterns in providing the awareness of vulnerabilities. We found that there is a great deal of hype, but very little hard evidence about the benefits and pitfalls of using anti-patterns in software engineering education and none on capturing vulnerabilities

from cybersecurity community (see Section 2.14). It is apparent from the findings of our studies that anti-patterns have the potential to help to find vulnerabilities during the SDLC and to provide them with solutions. Therefore, the results suggest that giving anti-patterns as developers' train have a significant effect on transferring vulnerability knowledge to software developers. Anti-patterns capture poor practices. Vulnerabilities are poor practices which can be exploited by hackers, so using anti-patterns to transfer poor security practice knowledge shows promise in reducing code related vulnerabilities.

9.2 Experimental Studies Limitations

9.2.1 Avoidance Bias

Several steps were taken to avoid introducing any type of bias into the experiments. When partaking in the experiments, participants were initially told the purpose of the research was to assess and measure their core understanding and mitigation knowledge of software developers about the most commonly occurring software errors (vulnerabilities). Participants were asked to complete a questionnaire and some simple tasks. All instructions were provided before each task began. No risks or discomforts were anticipated from taking part in this study. For data analysis and interpretation, participants were asked to answer all the questions and complete the tests. However, if a participant felt uncomfortable with any of our questions/tests, he/she was able to skip that question/test or withdraw from the study altogether. Participation was voluntary. All data and responses were kept entirely confidential. The data did not contain any personal information except participants age, gender and degree programme. Participants' names were replaced with a participant number, and it is not possible to identify people from any of the gathered data.

9.2.2 Very Small Sample Size

Results obtained of studies such as PS-I, PS-II, and industrial studies were insignificant due to the very small sample size. In the PS-I, students were recruited regardless of their background knowledge, considering all of them to be software developers. The results of PS-I show that participants with a cybersecurity background are aware of vulnerable codes and not considered to

be software developers. Therefore, during PS-II we recruited only final year computing-related degree students, known as software developers. In PS-II, only those students were recruited who were well versed in specific programming languages. Only 38 students were recruited in PS-II.

Professional programmers are hard to recruit and are expensive. During the Industrial study, only five professional software developers were recruited. Furthermore, it is difficult to come by an adequate pool of professional developers in locations that do not have a significant software developmental industrial base. In PS-I, participants with cybersecurity knowledge results were not reliable. To rectify these unreliable results, during PS-II, we only recruited computing related degree students known as software developers. However, during Industrial study, one of the participant's possessed cybersecurity knowledge, which influenced our results.

9.2.3 Students as Participants

During our research, it was relatively easy to use students as subjects in PS-I and PS-II. Such experiments in software engineering are often criticized due to their artificial settings. To prove the practical implementation of experimental results in a realistic setting, the industrial study included sample subjects from the professional developer population that we aim to make claims about. Students are more accessible, easier to hire and are generally inexpensive. PS-I and PS-II were easier to run and to reduce the risks, than experiments with professionals. PS-I and PS-II were carried out to test experimental design and initial hypotheses with students as participants, before conducting experiments with professionals' developers. The potential goal of PS-I and PS-II was to gain an understanding of the basic issues without deliberately aiming for external validity. Conducting experiments with students was the first step to the measured effectiveness of VAPs in the knowledge transfer process, and the aim was to reduce risks and costs. However, students lack the experience or understanding of professionals, which is the main disadvantage of PS-I and PS-II. This affects the validity of results. In our research, the students were paid, and the experiment was not considered a part of the course. PS-I and PS-II were carried out to test experimental design and initial hypotheses with students as participants, before conducting experiments with professional developers.

9.2.4 Background Knowledge of Subjects

Background knowledge of participants was another important factor that influenced the experimental results. On the basis of participants' background knowledge, they were divided into two groups: ethical hackers, and software developers (PS-I). Ethical hackers possessed knowledge of cybersecurity that influenced experimental results. Hence during PS-II, ethical hacking students were not recruited as potential participants. Consequently, the initial number of potential participants was low. Moreover, the remaining participants were only software developers, who were assumed to lack the awareness of errors that led to vulnerabilities. In addition, the remaining participants had learnt different programming modules. For example, gaming students were aware of C\C++ programming languages, and computing students were aware of PHP programming languages. Thus, participants were further sub-divided that reduced the sample size and made it difficult to get statistically significant results.

9.2.5 Lack of Realistic Environment

Experiments were executed in an artificial environment, with a controlled group and in a short period of time. This makes it difficult to obtain meaningful results.

9.2.6 Evaluating usability

This research carried out three main studies to evaluate the effectiveness of VAPs in order to provide developers with an understanding of code vulnerabilities. In PS-II, control and treatment (experimental) groups' comparison suggest that participants improved their ability to understand and mitigate vulnerabilities via VAPs. The scope of this study only considered the software engineering and cybersecurity domain aspects related to vulnerabilities. Due to limited research time, these studies were unable to verify usability, which requires a longitudinal field experiment while considering the other influential factors such as participant psychological and behavioural variables. To verify the efficacy of VAPs, we plan to carry out a longitudinal field quasi-experiment. This is a part of our future research (see Section 10.4.3).

10 Conclusion and Future Work

This thesis discussed the transfer of desirable and feasible cybersecurity domain knowledge from security experts ("Ethical Hackers") to software engineers. The mechanism of knowledge transfers between the work on vulnerability databases (VDBs), developers' perceptions of security issues and the security development lifecycle (SDLC) is complex, which creates a distinct communication gap between ethical hackers and software engineers as mentioned in Section 2.11. This security knowledge gap prevents software developers from making use of security domain knowledge in its form of vulnerability databases (e.g. CWE, CVE, Exploit DB), which are therefore not appropriate for this purpose. The identification of these problems provided the motivation and requirement for a useful technique for transferring vulnerability knowledge. A solution is proposed in Section 3.4 Vulnerability Anti-Pattern that based upon the improved use of anti-pattern, which encompasses security domain knowledge. A catalogue of Vulnerability Anti-Patterns (see Section 5.4) is developed to provide developers with an effective understanding of poor security practices that lead to vulnerabilities. A series of experimental studies (see evaluation Section) has been performed to validate the proposed hypothesis (see Section 1.8). The results highlight that Vulnerability Anti-Pattern appears to be of value in providing an effective understanding of vulnerabilities.

This chapter concludes the thesis by relating the results and contributions to the hypothesis in subsequent Section 10.1. Possible further work is summarised in Section 10.5.

10.1 Conclusion

A hybrid pattern-based technique to capture security knowledge during development has been defined. The overall goal of this thesis is to contribute to the development of secure software systems by improving security vulnerabilities awareness among developers:

- **Increasing the Security Flaws Awareness among Developers.** A series of case studies performed for the following purposes: to raise awareness how vulnerabilities could be exploited by malicious hackers, to

measure the effectiveness of VAPs for developers in raising the awareness of poor security practices that lead to vulnerabilities. The significant results were concluded through the qualitative and quantitative research that VAPs are significantly useful in increasing vulnerabilities awareness among developers.

- **Awareness of Vulnerabilities via Vulnerability Anti-Pattern (VAP).** Vulnerability Anti-Pattern technique can be used to provide developers awareness about how a malicious attacker can exploit a security flaw and misuse a system. Each VAP provides insight for developers to think about security systematically using an anti-pattern to signify poor security practices and a pattern to provide mitigation to solve the identifiable anti-pattern. The contribution of this thesis is to help developers in identifying anti-patterns (vulnerabilities) and enable them to use patterns (solutions/ mitigations) against vulnerabilities.
- **Bridging the Security Knowledge Gap between Cybersecurity Experts and Software Developers.** VAP encapsulates vulnerability information from cybersecurity experts and presents a format that can be utilised by developers; anti-pattern (i.e. poor security practices) and pattern (i.e. mitigation and solutions mapping into SDLC) template. This enables tighter coupling between cybersecurity expertise and software development practice to bridge the knowledge gap. A series of experiments was performed to measure this; however, further evaluation of VAP effectiveness will be explored in our future work.

Since new security attacks are being discovered and launched all the time, the security solutions to prevent the attacks will need to change, and so will need for the corresponding Vulnerability Anti-Pattern to stop security attacks.

10.2 Primary Contribution: Vulnerability Anti-Pattern and its Catalogue: A Timeless Way to Capture Poor Software Practices (Vulnerabilities)

To mitigate vulnerabilities, we anticipated a novel approach called Vulnerability Anti-Pattern that helps developers understand vulnerabilities, coupled with how to mitigate them during software development practices. We propose an

extended template of anti-patterns and using this template we produce a catalogue of VAPs against 12 vulnerabilities (Table 83 Catalogue of Vulnerability Anti-Patterns), chosen from the OWASP list of “Top 25 Most Dangerous Software Errors”. Our current catalogue covers SANS Top 10 most commonly occurring software errors in “Informal” and “Formal” format for today’s developers. We plan to continue maintaining and extending our catalogue in future.

#	**Vulnerability Name (sourced from CWE (MITRE Corporation 2015e))**	**Vulnerability Anti-Pattern**
1	Improper Neutralization of Special Elements used in an SQL Command	SQL Injection
2	Missing Authentication	Missing Authentication for Critical Functions
3	Missing Authorization	Missing Authorization
4	Buffer Copy without Checking Size of Input	Buffer Overflow
5	Use of Obsolete Function	Use of Deprecated Function
6	Use of Potentially Dangerous Function	Use of Potentially Dangerous Function Calls
7	Integer Overflow or Wraparound	Integer Overflow
8	Incorrect Calculation of Buffer Size	Incorrect Calculation of Buffer Size
9	Improper Neutralization of Input During Web Page Generation	Cross-Site Scripting (XSS)
10	Cross Site Reference Forgery	Cross-Site Request Forgery
11	Use of Externally-Controlled Format String	Format String Injection
12	Shell injection	OS Command Injection

Table 83 VAPs catalogue

Despite the relative success of the VAP addressing the fundamental problem of cybersecurity (vulnerabilities), it only provides a small contribution to solving the large research problem of aiming to provide developers awareness of eleven from thousands of discovered vulnerabilities. Future work will continue to extend this catalogue and will cover all possible sets of vulnerabilities.

10.3 Research Outcomes

The essential argument made in the thesis statement (Section 1.6) is that developing an awareness of poor software engineering practices through the use of a pattern-based approach can help software developers in the creation of more secure software by communicating recurrent exploitable software errors repeatedly made during the SDLC. However, existing pattern-based approaches are limited by their complicated structure and a lack of understanding of the genesis of development errors (Section 2.13) that lead to vulnerabilities.

The research question driving this research was "*Can a pattern-based approach (Vulnerability Anti-Pattern) be effective to fill the security knowledge gap between software developers and security experts in order to help developers in the creation of secure software systems?*" We divided the research into the following questions:

- **RQ1:** Do software developers have an effective understanding of errors that lead to the creation of vulnerabilities, coupled with an awareness of how malicious hackers can exploit these errors?
    - In response to RQ1, this research demonstrates that developers lack an effective understanding of recurrent vulnerabilities as investigated in the literature review (Chapters 2 and 3) and evaluated through experimental studies (Chapters 6,7 and 8).
- **RQ2:** Why are current attempts, in the form of patterns and catalogues of vulnerabilities, not successful in communicating security knowledge to software developers?
    - In response to RQ2, Current attempts in the form of patterns and catalogues of vulnerabilities, are not successful in communicating security knowledge to developers as explored through literature of existing pattern-based approaches (Chapters 2 and 3) and proposed a solution: Vulnerability Anti-Pattern, which based on improved use of pattern-based approaches to capture poor software development practices (vulnerabilities) (Chapters 4 and 5).

- **RQ3:** Do developers know how to mitigate these recurrent errors during the Software Development Lifecycle (SDLC)?
    - In response to RQ3, a series of experiments were performed with (students and professional software developers) participants to explore that do developers know how to mitigate these recurrent errors during the Software Development Lifecycle (SDLC).

We address RQ1 and RQ2 in Chapter 2 and 3, by analysing existing pattern-based approaches and evaluating the issues why current attempts are not successful. This thesis addresses these issues by proposing "Vulnerability Anti-Pattern (VAP)" that assists software developers in developing an awareness of how malicious hackers can exploit errors in software (Chapter 4). Furthermore, Anti-Patterns can provide sufficient awareness of vulnerabilities in order to enable developers to create more secure software (RQ1). This bridges the security knowledge gap between software engineers and cybersecurity communities by providing information about cybersecurity issues in formats that are usable and understandable for developers.

- In response to the main research question, Vulnerability Anti-Pattern can help developers to improve their awareness of recurrent vulnerabilities.

We perform a series of experiments in the evaluation section of this dissertation (Chapters 6, 7 and 8), which was based on a series of experimental studies to measure VAPs effectiveness in raising professional software developers' awareness of common software vulnerabilities.

- Quantitative analysis of **Pilot-study-I** suggests that computing degree related students (Developers) were not able to identify recurring security flaws and were not able to demonstrate an understanding of how malicious hackers can exploit these flaws. In comparison, the ethical hacking students performed well in being able to identify recurring security flaws and demonstrating an understanding of how these can be exploited.
- Quantitative analysis of **Pilot-study-II** shows that VAPs as an intervention, provide developers with the understanding and awareness of poor security practices (vulnerabilities) during SDLC,

however, there is no significant difference between “Informal” and “Formal” Vulnerability Anti-Pattern. The most interesting finding was that an “informal intervention was more effective in order to help participants to retain an awareness of vulnerabilities after one week gap” in comparison to formal intervention.

- Qualitative analysis of the **Industrial study** shows that intervention through VAPs help participants to understand and identify the root-cause of vulnerabilities.

10.4 Significance of the Research

Vulnerability Anti-Pattern is a new pattern-based approach to identify and organise vulnerabilities that can provide support for developers to prevent and mitigate vulnerabilities. Results from a series of experimental studies recommend that Vulnerability Anti-Pattern is an appropriate way to provide vulnerability awareness training about poor security practices. The work of this thesis is concentrated on Vulnerability Anti-Pattern as a solution against recurring vulnerabilities in a reusable and understandable format for developers’ aspects, and there is much to do beyond these narrow boundaries. The following section will describe some future directions for our work and project our vision.

10.5 Future Work

There is a potential to build on this current work as well as expand it into newer terrains. Proposed here are some research areas that could be exploited in future:

10.5.1 Catalogue of Vulnerability Anti-Patterns

The Vulnerability Anti-Pattern (VAP) catalogue will grow in response to new classes of vulnerabilities and proactively, as more people use it and begin to look for missing security flaws. An automated tool will generate to keep the databased up-to-date. A developer can use multiple VAPs as a reference guide to prevent vulnerabilities during software development processes. Identify the connection among different sets of vulnerabilities and their mapping into SDLC. Furthermore, exploring the relationship between VAPs help us to understand how VAPs can be composed against interlinked vulnerabilities.

Furthermore, it is possible to study a particular programming language or platforms and identify appropriate Vulnerabilities Anti-Patterns. This research project is particularly interested in concentrating on vulnerabilities that mistakenly occurred due to a developer's mistake.

10.5.2 Design a Pattern Language

Security threats evolve rapidly, requiring developers to be up to date with the latest information in order to prepare appropriately for software attacks. Education of security for developers is an on-going process. As new vulnerabilities will discover, more VAPs will generate against them. There is a need to design a pattern-language in order to classify the vulnerabilities with their vulnerable exploitation patterns, which will benefit developers to identify, prevent and predict potential interlinked or dependent vulnerabilities. My next step is to design a pattern-language based on to classify VAPs.

10.5.3 VAPs Evaluation Considering Usability and Retainability

In this research, VAPs was evaluated to provide an understanding of vulnerabilities. There are many aspects related to VAPs evaluation, which are essential to consider such as usability, efficiency and learnability. However, these are out of the scope of our research. Nevertheless, this is my future plan to examine these above factors and would run trials with a significant number of professional developers before developing a commercial product.

10.5.4 Vulnerability Anti-Pattern Tool

Vulnerability Anti-Pattern (VAP) has captured vulnerability information as depicted in a pattern and anti-pattern, which can be utilised to generate an automated security tool. I am aiming to build a tool based on VAP, which would warn developers of vulnerabilities during real-time development practices.

10.5.5 Training Method to Educate Developers about Recurrent Vulnerabilities

Software systems should be as secure as they are usable, but threats to, and vulnerabilities within, the augmented complex network of people and software

systems make this a challenging task for software developers. Education of security through an Anti-Pattern is a beneficial mechanism.

I am aiming to design a security training package which will use the catalogue of Vulnerability Anti-Patterns. This awareness training can help developers to learn from others mistakes and gives insight on how to prevent them. This training will include in-house and online security training sessions to provide developers with awareness about the most-up-to-date security flaws that cause vulnerabilities.

As a result, software developers can able to apply proactive and robust security measures, delivering secure software products that are not an easy target for cyber-attacks.

10.5.6 VAP Catalogue Dissemination

For academic purposes, the online directory will be created to publicise the VAPs. However, for commercialisation, VAPs will be used to develop a security training and automated tool. Therefore, VAPs will be publicised under copyright protection for academic researchers to use. The authors are collaborating with the Pattern Languages of Programs (PLoP) researchers to extend and disseminate the VAP catalogue for educational purposes.

11 References

Acar, Y. et al. 2016. You get where you're looking for: The impact of information sources on code security. In: *IEEE Symposium on Security and Privacy*, San Jose, CA, 22-26 May 2016. IEEE, pp.289-305.

Alexander, R., D. and Panguluri, S. 2017. Cybersecurity terminology and frameworks. In: R. M. Clark and S. Hakim. eds. *Cyber-physical security: Protecting critical infrastructure at the state and local level.* Cham: Springer. pp.19-47.

Alexander, C., Ishikawa, S. and Silverstein, M. 1977. *A pattern language.* Available from: http://courses.cs.vt.edu/~cs4634/fall2004/lecturehandouts/designpatterns.pdf [Accessed 23 October 2014].

Allen, J., et al., 2001. *Code red worm exploiting buffer overflow in IIS Indexing Service DLL.* Technical Report. Carnegie Mellon Software Engineering Institute.[Online] Available from: https://resources.sei.cmu.edu/asset_files/WhitePaper/2001_019_001_496466.pdf [Accessed February 2016].

Alvarez, M. et al. 2017. IBM X-Force Threat Intelligence Index 2017 The Year of the Mega Breach. *IBM Security.* (March): pp.1-30.

Alvi, A.K. and Zulkernine, M. 2011. A natural classification scheme for software security patterns. In: *Ninth International Conference on Dependable, Autonomic and Secure Computing*, 12-14 Dec. 2011. IEEE, pp.113-120.

Amoroso, E. G. 1994. *Fundamentals of computer security technology.* London: Prentice Hall International.

Anand, P., Jungwoo Ryoo, R. and Kazman, R. 2014. Vulnerability-based security pattern categorization in search of missing patterns. In: *Ninth International Conference on Availability, Reliability and Security*, 8-12 Sept. 2014. IEEE, pp.476-483.

Anastas, J. W. 1999. *Research design for social work and the human services.* New York: Columbia University Press.

Arnold, A.D., Hyla, B.M. and Rowe, N.C. 2006. Automatically building an information-security vulnerability database. In: *IEEE Information Assurance Workshop*, 21-23 June 2006. West Point, pp.376-377.

Arshad, J. et al. 2012. Cloud computing security: Opportunities and pitfalls. *International Journal of Grid and High Performance Computing (IJGHPC).* 4(1): pp.52-66.

Aslam, T., Krsul, I. and Spafford, E.H., 1996. *Use of a taxonomy of security faults.* Purdue University. [Accessed 12 October 2015].

Banerjee, C. and Pandey, S. K. 2009. Software Security Rules, SDLC Perspective. *(IJCSIS) International Journal of Computer Science and Information Security.* 6(1): pp.123-128.

Beizer, B. 2003. *Software testing techniques.* 2nd ed. New Delhi: Dreamtech Press.

Beizer, B. 1990. *Software testing techniques.* 2nd ed. New York: Van Nostrand Reinhold.

Bekrar, S. et al. 2011. Finding software vulnerabilities by smart fuzzing. In: *Fourth IEEE International Conference on Software Testing, Verification and Validation,* 21-25 March 2011. IEEE, pp.427-430.

Benedikt, M. 1990. Cyberspace: Some proposals. In: *Conference on Cyberspace*, May 1990. University of Texas, pp.5-6.

Black, P. E. 2017. *SARD: A Software Assurance Reference Dataset.* Available from: https://samate.nist.gov/index.php/SARD.html[Accessed May 2016].

Blackwell, C. and Zhu, H. 2014. Future directions for research on cyberpatterns. In: C. Blackwell and H. Zhu. eds. *Cyberpatterns: Unifying design patterns with security and attack pattern.* Springer. pp.259-264.

Blackwell, C. and Zhu, H. 2014. *Cyberpatterns: Unifying design patterns with security and attack patterns.* Heidelberg: Springer.

Board, I. 1993. IEEE standard classification for software anomalies. *IEEE Standard.* 1044.

Borstad, O. G. 2008. *Finding security patterns to countermeasure software vulnerabilities.* Master Thesis. Institutt for Datateknikk og Informasjonsvitenskap.

Bourque, P. and Fairley, R. E. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press.

Brehmer, C. and Carl, J. 1993. Incorporating IEEE Standard 1044 into your anomaly tracking process. *CrossTalk, Journal of Defense Software Engineering.* 6: pp.9-16.

Brenner, J. 2007. ISO 27001: Risk management and compliance. *Risk Management.* 54(1): pp.24-29.

Brown, W. J., 1998. *AntiPatterns: Refactoring software, architectures, and projects in crisis.* New York: Wiley.

Bunke, M. 2015. Software-security patterns: Degree of maturity. In: *Proceedings of the 20th European Conference on Pattern Languages of Programs*, July 08 - 12, 2015 2015. ACM, pp.42-61.

Bunke, M., Koschke, R. and Sohr, K. 2012. Organizing security patterns related to security and pattern recognition requirements. *International Journal on Advances in Security.* 5(2): pp.46-67.

Burnstein, I. 2006. *Practical software testing: A process-oriented approach.* New York: Springer Science & Business Media.

Busch, M., Koch, N. and Wirsing, M. 2014. Evaluation of engineering approaches in the secure software development life cycle. In: M. Heisel, et al. eds. *Engineering secure future internet services and systems.* Springer. pp.234-265.

C/S2ESC - Software & Systems Engineering Standards Committee, 2010. *1044-2009-IEEE Standard Classification for Software Anomalies.* IEEE. [Accessed 12 August 2017].

Cabinet Office, 2011. *The cost of cyber crime.* Cabinet Office Report. [Accessed 10/12/2014].

Cabinet Office, 14 April 2016. *The UK cyber security strategy report on progress and forward plans.* UK: Cabinet Office and National security and intelligence. [Accessed 2015].

Calcutt, A. 1999. *White noise: An A–Z of the contradictions in cyberculture.* London: Palgrave Macmillan.

Carnegie Mellon University, 2015. *The Vulnerability Notes Database.* U.S.A.: CERT, Carnegie Mellon University. [Accessed 12 November 2014].

CERT, 2015. *Common Cyber Security Language - ICS-CERT - US-CERT.* US-CERT. [Accessed 01 January 2016].

Chen, Z., Zhang, Y. and Chen, Z. 2010. A categorization framework for common computer vulnerabilities and exposures. *The Computer Journal.* 53(5): pp.551-580.

Cherdantseva, Y. and Hilton, J. 2014. Information security and information assurance: Discussion about the meaning. In: I. M. Portela and F. Almeida. eds. *Organizational, legal, and technological dimensions of information system administration.* Hershey, PA: IGI Global. pp.1204-1235.

Cheswick, W. R., Bellovin, S. M. and Rubin, A. D. 2003. *Firewalls and internet security: Repelling the wily hacker.* Boston. MA: Addison-Wesley Longman Publishing Co.

Choo, K. R. 2011. The cyber threat landscape: Challenges and future research directions. *Computers & Security.* 30(8): pp.719-731.

Cockburn, A. and Baruz, A. 2017. *Antipattern.* Available from: http://wiki.c2.com/?AntiPattern [Accessed August 2015].

Cohen, F. 1999. Simulating cyber attacks, defences, and consequences. *Computers & Security.* 18(6): pp.479-518.

Committee on National Security Systems (CNSS), 2003. *National Information Assurance Glossary.* 4009. CNSS.[Online] Available from: https://www.ecs.csus.edu/csc/iac/cnssi_4009.pdf [Accessed 20 August 2017].

Conte, S. D., Dunsmore, H. E. and Shen, V. Y. 1986. *Software engineering metrics and models.* US: Benjamin-Cummings Publishing.

Conti, G. and Caroland, J. 2011. Embracing the Kobayashi Maru: Why you should teach your students to cheat. *IEEE Security & Privacy.* 9(4): pp.48-51.

Craigen, D., Diakun-Thibault, N. and Purse, R. 2014. Defining cybersecurity. *Technology Innovation Management Review.* 4(10): pp.13-21.

Csanadi, L. C. G. 2015. Cyber war: Poor man's weapon of mass destruction, and a new whip in the hands of the rich. *Defence.* 143: pp.153-174.

Daud, M.I. 2010. Secure software development model: A guide for secure software life cycle. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*, March 17-19 2010., pp.978-988.

De Win, B. et al. 2009. On the secure software development process: CLASP, SDL and Touchpoints compared. *Information and Software Technology.* 51(7): pp.1152-1171.

Department of Homeland Security (DHS), 2017. *Cyber Incident Response at DHS.* 2017. USA: DHS.[Online] Available from: https://www.us-cert.gov/sites/default/files/publications/infosheet_SoftwareAssurance.pdf [Accessed 01 January 2017].

Dias e Silva, P. 2014. *Categorization of Anti-Patterns in Software Project Management.* Master Thesis. Universidad Politecnica de Madrid.

Dimitrov, V., 2016. *Toward formalization of software security issues.* Bulgaria: University of Sofia. [Accessed 02 September 2016].

Din, J., Al-Badareen, A. and Jusoh, Y.Y. 2012. Antipatterns detection approaches in object-oriented design: A literature review. In: *7th International Conference on Computing and Convergence Technology (ICCCT).*, 3-5 Dec. 2012. Seoul, South Korea: IEEE, pp.926-931.

Dougherty, C.R., et al., 2009. *Secure design patterns.* Carnegie Mellon University: Defense Technical Information Center.[Online] Available from: http://www.dtic.mil/docs/citations/ADA636498 [Accessed August 2015].

Druin, J. 2011. *OWASP Mutillidae II Web Pen-Test Practice Application.* Available from: https://sourceforge.net/projects/mutillidae/ [Accessed 13 June 2016].

Fahl, S. et al. 2013. Rethinking SSL development in an appified world. In: *Proceedings of the ACM Conference on Computer and Communications Security*, November 04 - 08 2013. ACM, pp.49-60.

Faily, S., Parkin, S. and Lyle, J. 2014. Evaluating the implications of attack and security patterns with premortems. In: C. Blackwell and H. Zhu. eds. *Cyberpatterns: Unifying design patterns with security and attack patterns.* Cham: Springer. pp.199-209.

Fal', A. 2010. Standardization in information security management. *Cybernetics and Systems Analysis.* 46(3): pp.512-515.

Fernandez, E.B., Yoshioka, N. and Washizaki, H. 2010. A worm misuse pattern. In: *Proceedings of the 1st Asian Conference on Pattern Languages of Programs*, March 16 - 17 2010. ACM, pp.1-5.

Fernandez, E.B., Yoshioka, N. and Washizaki, H. 2009. Modeling misuse patterns. In: *2009 International Conference on Availability, Reliability and Security*, 16-19 March 2009. IEEE, pp.566-571.

Fernandez, E.B. et al. 2008. Classifying security patterns. In: *Asia-Pacific Web Conference APWeb 2008*, 26-28 April 2008. Heidelberg: Springer, pp.342-347.

Fink, A. 2003. *The survey handbook.* London: Sage.

Fittkau, F. 2011. Controlled experiments in software engineering. In: *Seminar Empirical Methods SuSe 2011* 2011. Kiel: Department of Computer Science, Kiel University.

Flamm, K. 1988. *Creating the computer: Government, industry, and high technology.* Washington DC: Brookings Institution Press.

Foote, B., Rohnert, H. and Harrison, N. 1999. *Pattern languages of program design 4.* Boston, MA: Addison-Wesley Longman Publishing.

Foote, B. and Yoder, J. 1997. *Big ball of mud.* Available from: http://ansymore.uantwerpen.be/system/files/uploads/courses/SE3BAC/p11_1_bigballofmud.pdf.

Gamma, E., et al., 2008. *Design patterns: elements of.* Department of Computer Science, The Australian National University: Addison-Wesley.[Online] Available from: http://courses.cecs.anu.edu.au/courses/archive/comp2110.2008/lectures/lec16_4up.pdf [Accessed October 2015].

Garcia, M. et al. 2014. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience.* 44(6): pp.735-770.

Garfinkel, S., Spafford, G. and Schwartz, A. 2003. *Practical UNIX and internet security.* Sebastopol, CA: O'Reilly Media.

Ghani, H. et al. 2013. *International Conference on Risks and Security of Internet and Systems (CRiSIS)*, 23-25 Oct. 2013. , pp.1-8.

Green, M. and Smith, M. 2016. Developers are not the enemy! : The need for usable security APIs. *IEEE Security & Privacy.* 14(5): pp.40-46.

Gregoire, J. et al. 2007. On the secure software development process: CLASP and SDL compared. In: *Third International Workshop on Software Engineering for Secure Systems*, 20-26 May 2007. IEEE, pp.1-1.

Gu, Y. and Li, P. 2010. Design and research on vulnerability database. In: *Third International Conference on Information and Computing*, 4-6 June 2010. Wuxi, PR China, pp.209-212.

Guttman, B. 1995. *An introduction to computer security: The NIST handbook.* Gaithersburg, MD: Dept. of Commerce, Technology Administration, National Institute of Standards and Technology.

Hafiz, M. 2011. *Security on demand.* Doctoral dissertation. University of Illinois at Urbana-Champaign.

Halkidis, S. T., Chatzigeorgiou, A. and Stephanides, G. 2006. A qualitative analysis of software security patterns. *Computers & Security.* 25(5): pp.379-392.

Hans, K. 2010. Cutting Edge Practices for Secure Software Engineering. *International Journal of Computer Science and Security IJCSS.* 4(4): pp.403-408.

Happel, J. 2017. *Web Application Penetration Testing With Burp Suite.* Available from: https://www.pluralsight.com/courses/web-application-penetration-testing-with-burp-suite [Accessed October 2017].

Heim, M. 1991. The erotic ontology of cyberspace. In: M. Benedikt. ed. *Cyberspace: First steps.* Cambridge: MIT Press. pp.59-80.

Höst, M., Wohlin, C. and Thelin, T. 2005. Experimental context classification: Incentives and experience of subjects. In: *Proceedings of the 27th International Conference on Software Engineering* 2005. ACM, pp.470-478.

Howard, M. and Lipner, S. 2011. *The security development lifecycle.* 2nd ed. The University of California: Microsoft Press.

Howard, M. and Lipner, S. 2009. *The security development lifecycle: SDL: A process for developing demonstrably more secure software.* 2nd ed. California: Microsoft Press.

Howard, M. and Lipner, S. 2006. *The security development lifecycle: A process for developing demonstrably more secure software.* The University of California: Microsoft Press.

Howard, M. 2005. How do they do it?-A look inside the security development lifecycle at Microsoft. *MSDN Magazine.* (November): pp.107-114.

Howard, M. 2004. Building more secure software with improved development processes. *IEEE Security & Privacy.* (6): pp.63-65.

Huang, C. et al. 2013. A novel approach to evaluate software vulnerability prioritization. *Journal of Systems and Software.* 86(11): pp.2822-2840.

Hui, Z. et al. 2010. Review of software security defects taxonomy. In: *International Conference on Rough Sets and Knowledge Technology*, 15-17 October 2010. Beijing, China, pp.310-321.

IEEE 1994. *IEEE standard classification for software anomalies. (1044-1993).* IEEE.

IEEE 1990. *IEEE standard glossary of software engineering terminology (610.12-1990).* Piscataway: IEEE.

Ilyin, Y. 2015. *Can we beat software vulnerabilities?* Available from: https://business.kaspersky.com/can-we-beat-software-vulnerabilities/2425 [Accessed Aug 22, 2014].

Jafari, A.J. and Rasoolzadegan, A. 2016. Securing gang of four design patterns. In: *Proceedings of the 23rd Conference on Pattern Languages of Programs*, October 24 - 26 2016. The Hillside Group, pp.5.

Jang-Jaccard, J. and Nepal, S. 2014. A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences.* 80(5): pp.973-993.

Jorgensen, P. C. 2013. *Software testing: A craftsman's approach.* New York, NY: CRC press.

Julisch, K. 2013. Understanding and overcoming cyber security anti-patterns. *Computer Networks.* 57(10): pp.2206-2211.

Juristo, N. and Moreno, A. M. 2013. *Basics of software engineering experimentation.* Boston: Springer Science & Business Media.

Kalaimannan, E. and Gupta, J. N. 2017. The security development lifecycle in the context of accreditation policies and standards. *IEEE Security & Privacy.* 15(1): pp.52-57.

Kidwell, P. A. 1998. Stalking the elusive computer bug. *IEEE Annals of the History of Computing.* 20(4): pp.5-9.

Kis, M. 2002. Information security antipatterns in software requirements engineering. In: *9th Conference of Pattern Languages of Programs (PloP)*, September 8th-12th 2002. PLoP, pp.1-7, Volume 11.

Kissel, R., 2013. *NIST IR 7298 Revision 2: Glossary of Key Information Security Terms.* 7. Gaithersburg, MD: National Institute of Standards and Technology (NIST). [Accessed July 2016].

Koenig, A. 1995. Patterns and antipatterns. *Journal of Object-Oriented Programming.* 8(1): pp.46-48.

Kotzé, P., Renaud, K. and Van Biljon, J. 2008. Don't do this–Pitfalls in using anti-patterns in teaching human–computer interaction principles. *Computers & Education.* 50(3): pp.979-1008.

Kotzé, P. et al. 2006. Patterns, anti-patterns and guidelines–effective aids to teaching HCI principles. In: *Inventivity: Teaching Theory, Design and Innovation in HCI, Proceedings of of HCIEd2006-1 First Joint BCS/IFIP WG13. 1/ICS/EU CONVIVIO HCI Educators' Workshop, 23-24 March 2006, Limerick, Ireland* 2006. Citeseer, pp.109-114.

Krsul, I. V. 1998. *Software vulnerability analysis.* PhD. Purdue University.

Lee, E.A. 2008. Cyber physical systems: Design challenges. In: *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, May 05 - 07 2008. IEEE, pp.363-369.

Leveson, N. 2004. A new accident model for engineering safer systems. *Safety Science.* 42(4): pp.237-270.

Levi, M., et al., 2016. *The implications of economic cybercrime for policing.* City of London Corporation.[Online] Available from:

http://orca.cf.ac.uk/88156/1/Economic-Cybercrime-FullReport.pdf [Accessed January 2016].

Levy, S. 2001. *Hackers: Heroes of the computer revolution.* Garden City: Penguin Books.

Li, F. et al. 2017. Cyberspace-oriented access control: Model and policies. In: *IEEE Second International Conference on Data Science in Cyberspace (DSC),* 26-29 June 2017. IEEE, pp.261-266.

Lightsein, H. 2008. *First Recorded Usage of "Hacker".* Available from: https://manybutfinite.com/post/first-recorded-usage-of-hacker/ [Accessed 30 September 2017].

Liu, Q. and Zhang, Y. 2011. VRSS: A new system for rating and scoring vulnerabilities. *Computer Communications.* 34(3): pp.264-273.

Lohr, S. and Markoff, J. 2006. Windows is so slow, but why. *The New York Times.* [online]. March 27, 2006.

Lomont, C. C. and Jacobus, C. J. 2014. *System and methods for detecting software vulnerabilities and malicious code.* Google Patents.(8,806,619) [Online] Available from: https://patents.google.com/patent/US8806619B2/en [Accessed 30 October 2017] .

Long, J. 2001. Software reuse antipatterns. *ACM SIGSOFT Software Engineering Notes.* 26(4): pp.68-76.

Longstaff, T. A. et al. 1997. Security of the Internet. *The Froehlich/Kent Encyclopedia of Telecommunications.* 15: pp.231-255.

Loureiro, N. 2002. Programming PHP with security in mind. *Linux Journal.* 2002(102): pp.2-3.

Mansourov, N. 2011. *System assurance beyond detecting vulnerabilities.* Amsterdam: Elsevier/Morgan Kaufmann.

Mansourov, N. and Campara, D. 2010. Chapter 5 - knowledge of risk as an element of cybersecurity argument. In: N. Mansourov and D. Campara. eds. *System assurance beyond detecting vulnerabilities.* 1st ed. Boston: Morgan Kaufmann. pp.111-146.

Martin, B., et al., 2011. *2011 CWE/SANS top 25 most dangerous software errors.* 7515. The MITRE Corporation.[Online] Available from: http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf [Accessed November 2014].

Maymí, F. et al. 2018. Towards a definition of cyberspace tactics, techniques and procedures. In: *IEEE International Conference on Big Data*, 1-14 Dec. 2018. IEEE, pp.4674-4679.

McConnell, S. 1993. *Code complete: A practical handbook of software construction.* Redmond: Microsoft Press.

McGraw, G. 2012. Software Security. *Datenschutz Und Datensicherheit-DuD.* 36(9): pp.662-665.

McGraw, G. 2009. Software security. *Security & Privacy.* 2(2): pp.80-83.

McGraw, G. 2006. *Software security: Building security in.* Boston: Addison Wesley.

McLellan, V. 1981. Case of the purloined password: The F.B.I. wants to know who swiped an electronic file opening access to all sorts of data at a subsidiary of Dun and Bradstreet. *The New York Times.* [online]. July 26: pp.F4.

McMillan, R. 2012. The world's first computer password? It was useless too. *Wired.Com.* (January): pp.1-27.

Meyers, S. D. 2005. *Effective C++ : 55 specific ways to improve your programs and designs.* 3rd ed. London: Addison-Wesley.

Miller, B.P., et al., 1995. Technical Report CS-TR-1995-1268, University of Wisconsin.[Online] Available from: http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/fuzz-revisited.pdf [Accessed 9 November 2017].

MITRE Corporation 2016. *CVE-2016-8820.* Available from: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8820 [Accessed June 2016].

MITRE Corporation 2015. *Common Attack Pattern Enumeration and Classification (CAPEC).* Available from: https://capec.mitre.org/ [Accessed 20 June 2016].

MITRE Corporation 2015. *Common Vulnerabilities and Exposures (CVE).* Available from: https://cve.mitre.org/ [Accessed November 2014].

MITRE Corporation 2015. *Common Weakness Enumeration.* Available from: http://cwe.mitre.org/ [Accessed November 2014].

MITRE Corporation 2015. *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow').* Available from: http://cwe.mitre.org/data/definitions/120.html [Accessed 15 October 2014].

MITRE Corporation 2015. *Common Weakness Enumeration (CWE).* Available from: https://cwe.mitre.org/ [Accessed November 2014].

MITRE Corporation 2014. *Common Attack Pattern Enumeration and Classification.* Available from: http://capec.mitre.org/index.html [Accessed December 2014].

MITRE Corporation 2004. *A CVE Based Vulnerability Database.* Available from: https://www.cvedetails.com/ [Accessed November 2014].

Moral-García, S. et al. 2014. Enterprise security pattern: a new type of security pattern. *Security and Communication Networks.* 7(11): pp.1670-1690.

Morgan, S. 2016. *Is poor software development the biggest cyber threat?* Available from: http://www.csoonline.com/article/2978858/application-security/is-poor-software-development-the-biggest-cyber-threat.html [Accessed January 2015].

Mouratidis, H., Giorgini, P. and Manson, G. 2003. Integrating security and systems engineering: Towards the modelling of secure information systems. In: J. Eder and M. Missikoff. eds. *Advanced information systems engineering: International conference on advanced information systems engineering. Lecture notes in computer Science 2681.* Berlin: Springer. pp.63-78.

Murdico, V. 2007. *Bugs per lines of code.* Available from: http://amartester.blogspot.co.uk/2007/04/bugs-per-lines-of-code.html [Accessed 1 February 2015].

Nafees, T. et al. 2017. Idea-caution before exploitation: The use of cybersecurity domain knowledge to educate software engineers against software vulnerabilities. In: *Engineering Secure Software and Systems 9th International Symposium* 2017. Springer, pp.133-142.

National Cyber Security Centre, 2016. *National Cyber Security Strategy 2016.* HM Government.[Online] Available from: https://www.enisa.europa.eu/topics/national-cyber-security-strategies/ncss-map/national_cyber_security_strategy_2016.pdf [Accessed December 2016].

Newman, R. C. 2010. *Computer security : Protecting digital resources.* London: Jones and Bartlett Publishers.

Newton, N. 2010. The use of semi-structured interviews in qualitative research: strengths and weaknesses. *Exploring Qualitative Methods.* 1(1): pp.1-11.

Nielsen, J. 1993. *Usability engineering.* Revised ed. Heidelberg: Elsevier.

NIST 2015. *National Vulnerability Database.* Available from: https://nvd.nist.gov/ [Accessed 12 December 2015].

NIST 2011. *National Vulnerability Database (NVD).* Available from: https://nvd.nist.gov/ [Accessed December 2014].

NIST, 2011. *National Initiative for Cybersecurity Education Strategic Plan.* NIST.[Online] Available from: https://www.nist.gov/sites/default/files/documents/2017/04/14/nice-strategic-plan_sep2012.pdf [Accessed August 2015].

Orman, H. 2003. The Morris worm: A fifteen-year perspective. *IEEE Security & Privacy.* 99(5): pp.35-43.

OWASP 2015. *Buffer Overflows.* Available from: https://www.owasp.org/index.php/Buffer_Overflows [Accessed April 2015].

Pesante, L. 2002. *Encyclopedia of software engineering.* US: Wiley Online Library.

Petersen, K. and Wohlin, C. 2009. Context in industrial software engineering research. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement* 2009. IEEE Computer Society, pp.401-404.

Poller, A. et al. 2017. Can security become a routine?: A study of organizational change in an agile software development group. In: *ACM Conference on Computer Supported Cooperative Work and Social Computing*, February 25 - March 01 2017. , pp.2489-2503.

Radatz, J., Geraci, A. and Katki, F. 1990. IEEE standard glossary of software engineering terminology. *IEEE Std.* 610121990(121990): pp.3.

Raymond, E.,Steven, 2017. *A Brief History of Hackerdom.* 2017. Thyrsus Enterprises.[Online] Available from: https://immagic.com/eLibrary/ARCHIVES/GENERAL/AUTHOR_P/R000825P.pdf [Accessed 01 October 2017].

Riehle, D. 1997. Composite design patterns. *ACM SIGPLAN Notices.* 32(10): pp.218-228.

Robson, C. 2002. *Real world research: A resource for social scientists and practitioner-researchers.* 2nd ed. Oxford: Blackwell Publishing.

Runeson, P. et al. 2012. *Case study research in software engineering: Guidelines and examples.* Hoboken: John Wiley & Sons.

Russell, D. and Gangemi, G. 1991. *Computer security basics.* Cambridge: O'Reilly Media.

Russinovich, M. and Solomon, D. A. 2009. *Windows internals: Including windows server 2008 and windows vista.* WA, USA: Microsoft press.

Schumacher, M. et al. 2013. *Security patterns: Integrating security and systems engineering.* West Sussex: John Wiley & Sons.

Seacord, R. 2006. Secure coding in C and C of strings and integers. *IEEE Security & Privacy.* 4(1): pp.74-76.

Secunia 2015. *Secunia vulnerability database.* Available from: http://secunia.com/ [Accessed October 2014].

Severance, C. 2016. Bruce Schneier: the security mindset. *Computer.* 49(2): pp.7-8.

Shiralkar, T. and Grove, B., 2009. *Guidelines for Secure Coding.* OWASP. [Accessed January 2015].

Shostack, A., 2008. *Experiences threat modeling at Microsoft.* 413. Dept. of Computing, Lancaster University: Microsoft. [Accessed 12 September 2015].

Shull, F., Seaman, C. and Zelkowltz, M. 2006. Victor r. Basili's contributions to software quality. *IEEE Software.* 23(1): pp.16-18.

Sjoberg, D. I. K. et al. 2008. Building theories in software engineering. In: D. Sjøberg I. K., et al. . eds. *Guide to advanced empirical software engineering.* Heidelberg: Springer. pp.312-336.

Sommerville, I. 2010. *Software engineering.* New York: Addison-Wesley.

Stallings, W. et al. 2012. *Computer security: Principles and practice.* New Jersey,USA: Pearson Education.

Steel, C. and Nagappan, R. 2006. *Core security patterns: Best practices and strategies for J2EE", web services, and identity management.* India: Pearson Education.

Stroustrup, B. 1994. *The design and evolution of C.* India: Pearson Education.

Such, J. M. et al. 2016. Information assurance techniques: Perceived cost effectiveness. *Computers & Security.* 60: pp.117-133.

Sutter, H. and Alexandrescu, A. 2004. *C coding standards: 101 rules, guidelines, and best practices.* India: Pearson Education.

Sutter, H., 2002. *The New C : Smart (er) Pointers.* Dr. Dobb's Journal.[Online] Available from: http://www.drdobbs.com/cpp/the-new-csmarter-pointers/184403837 [Accessed December 2016].

Symantec Corporation, 2016. *2016 Internet Security Threat Report.* 04/16 21365088. Symantec Corporation.[Online] Available from: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf [Accessed 06 August 2016].

Symantec Corporation, 2008. *Internet Security Threat Report.* 23. USA: Symantec Corporation.[Online] Available from: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf [Accessed 23 April 2016].

The PHP Group, 2017. *Deprecated features in PHP 5.3.x.* The PHP Group.[Online] Available from: http://php.net/manual/en/migration53.deprecated.php [Accessed 11 November 2017].

Todorov, A. 2015. *User guide for open source project bug submissions.* Available from: http://opensource.com/business/13/10/user-guide-bugs-open-source-projects [Accessed 2013].

U.K National Crime Agency 2017. *National cyber crime unit (NCCU).* Available from: http://www.nationalcrimeagency.gov.uk/about-us/what-we-do/national-cyber-crime-unit [Accessed 14 December 2017].

Vacca, J. 2009. *Computer and information security handbook.* Waltham,MA: Morgan Kauffman.

Van Biljon, J. et al. 2004. The use of anti-patterns in human computer interaction: Wise or Ill-advised? In: *Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries* 2004. South African Institute for Computer Scientists and Information Technologists, pp.176-185.

Van Wyk, K. R. and McGraw, G. 2005. Bridging the gap between software development and information security. *Security & Privacy, IEEE.* 3(5): pp.75-79.

Von Solms, R. and Van Niekerk, J. 2013. From information security to cyber security. *Computers & Security.* 38: pp.97-102.

Walker, N. et al. 2014. A method for resolving security vulnerabilities through the use of design patterns. In: Anon. *Cyberpatterns.* Oxford: Springer. pp.149-155.

Walton, R. 2006. The Computer Misuse Act. *Information Security Technical Report; Information Security Technical Report.* 11(1): pp.39-45.

Weir, C., Rashid, A. and Noble, J., 2017. *Developer Essentials: Top Five Interventions to Support Secure Software Development.* Lancaster University. [Accessed August 2017].

Wichers, D., 2017. *OWASP Top-10 2017.* OWASP. [Accessed March 2017].

Wijayasekara, D., Manic, M. and McQueen, M. 2015. Vulnerability identification and classification via text mining bug databases. In: *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society.* 29 Oct-1 Nov. 2015. IEEE, pp.3612-3618.

Witschey, J. et al. 2015. Quantifying developers' adoption of security tools. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, August 31 - September 04 2015. , pp.260-271.

Wohlin, C. et al. 2014. *Experimentation in software engineering.* Heidelberg: Springer.

Wright, K. 2015. *6 truly shocking cyber security statistics.* Available from: https://www.itgovernance.co.uk/blog/6-truly-shocking-cyber-security-statistics/ [Accessed 16 August 2015].

Xiao, S., Witschey, J. and Murphy-Hill, E. 2014. Social influences on secure development tool adoption: Why security tools spread. In: *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, February 15 - 19 2014. , pp.1095-1106.

Xie, J., Lipford, H.R. and Chu, B. 2011. Why do programmers make security errors? In: *IEEE Symposium on Visual Languages and Human-Centric Computing*, 18-22 Sept 2011. IEEE, pp.161-164.

Yoshioka, N., Washizaki, H. and Maruyama, K. 2008. A survey on security patterns. *Progress in Informatics.* 5(5): pp.35-47.

Yskout, K., Scandariato, R. and Joosen, W. 2012. Does organizing security patterns focus architectural choices? In: *Software Engineering (ICSE), 2012 34th International Conference on*, June 02 - 09, 2012 2012. IEEE, pp.617-627.

Yskout, K., Scandariato, R. and Joosen, W. 2012. Does organizing security patterns focus architectural choices? In: *ICSE '12 Proceedings of the 34th International Conference on Software Engineering*, June 02 - 09 2012. , pp.617-627.

Yun-hua, G. and Pei, L., 2010. Design and research on vulnerability database. In: *ICIC 2010 - 3rd International Conference on Information and Computing,* 2010. IEEE, pp.209-212.